MASTER'S THESIS 2023

Evaluating a Unified Parallel Computing API for Radar Signal Processing

Filip Jergle Almquist

ISSN 1650-2884
LU-CS-EX: 2023-30

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-30

Evaluating a Unified Parallel Computing API for Radar Signal Processing

Utvärdering av ett universellt parallellt programmeringsgränssnitt för radarsignalbehandling

Filip Jergle Almquist

Evaluating a Unified Parallel Computing API for Radar Signal Processing

Filip Jergle Almquist
fi1316je-s@student.lu.se

June 27, 2023

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Anders Åhlander, anders.ahlander@saabgroup.com
Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Falvius Gruian, flavius.gruian@cs.lth.se

Abstract

This master's thesis presents an evaluation of Intel's oneAPI in the context of high-performance radar signal processing applications, with a focus on active electronically scanned array (AESA) radar systems. These systems require the real-time processing of large amounts of data, demanding high-performance hardware. However, the diversity of hardware platforms, including CPUs, GPUs, and FPGAs, introduces challenges for code portability and long-term maintenance. The study aims to assess oneAPI's ability to unify programming across these platforms, thereby improving engineering efficiency and sustainability.

The assessment centers on three main aspects: productivity, portability, and performance of oneAPI. The productivity evaluation showed that the amount of code required with oneAPI is comparable to other similar programming models. Portability investigations demonstrated that while oneAPI allows for code execution on multiple hardware architectures, code modifications are needed for optimized performance across devices, and experiments on FPGA were unsuccessful. The performance analysis indicated competitive results with other math libraries, but performance degradation was observed on third-party hardware.

The study thus concludes that oneAPI offers a step toward hardware-agnostic programming, although with issues such as bugs and limited industry adoption. It suggests that future work should extend the evaluation to other data-parallel algorithms and hardware types to fully understand oneAPI's potential in high-performance computing applications.

Keywords: oneAPI, Radar Signal Processing, Portability, High-Performance Computing, Parallel Computing

Acknowledgements

I would like to thank Anders Åhlander for teaching me everything I needed to know about radar systems and keeping the project on track.

I would also like to thank Jonas Skeppstedt for his academic guidance and the courses that let me dive deeper in the subject of high-performance computing.

Finally, I would like to thank Flavius Gruian for providing a new perspective on the thesis and for his helpful and extensive feedback.

Contents

1 Introduction **7**
1.1 Purpose . . . 8
1.2 Research Questions . . . 8
1.3 Scope . . . 8
1.4 Overview . . . 8

2 Radar Signal Processing **11**
2.1 Radar Introduction . . . 11
2.2 Signal Processing Pipeline . . . 12
2.3 Rader Signal Processing Realization . . . 13
2.3.1 Parallelization . . . 13
2.3.2 Accelerators . . . 14
2.3.3 CPU . . . 15

3 Programming Models **17**
3.1 oneAPI . . . 17
3.1.1 SYCL . . . 18
3.1.2 DPC++ . . . 18
3.1.3 Libraries . . . 18
3.1.4 Tools . . . 19
3.1.5 DevCloud . . . 19
3.2 Related Programming Models . . . 19
3.2.1 Kokkos . . . 19
3.2.2 Eigen . . . 20
3.2.3 CUDA . . . 20

4 Related Work **23**

5 Approach **25**
5.1 Productivity . . . 25

5.2 Portability . . . 25
5.3 Performance . . . 26
5.3.1 Comparison with Baseline . . . 26
5.3.2 Importance of Tuning Work Group Size . . . 26
5.3.3 Intel CPU vs Integrated GPU . . . 27
5.3.4 Discrete GPU . . . 27
5.3.5 CPU: oneAPI vs Kokkos . . . 28
5.3.6 GPU: oneAPI vs Kokkos vs CUDA . . . 28
5.3.7 Scaling Problem Size . . . 28
5.3.8 FPGA . . . 28
5.4 GPU Considerations . . . 28

6 Implementation **31**
6.1 Code Comparison . . . 31
6.1.1 DPC++ . . . 32
6.1.2 Kokkos . . . 33
6.1.3 CUDA . . . 34
6.2 Productivity Results . . . 35

7 Performance Results **37**
7.1 Comparison with Baseline . . . 37
7.2 Importance of Tuning Work Group Size . . . 38
7.3 Intel CPU vs Integrated GPU . . . 38
7.4 Discrete GPU . . . 38
7.5 CPU: oneAPI vs Kokkos . . . 39
7.6 GPU: oneAPI vs Kokkos vs CUDA . . . 39
7.7 Scaling Problem Size . . . 39

8 Discussion **47**
8.1 Interpretation of Results . . . 47
8.1.1 Productivity . . . 47
8.1.2 Portability . . . 47
8.1.3 Performance . . . 48
8.2 Broader Impact . . . 49
8.3 Limitations . . . 50
8.4 Future Work . . . 50

9 Conclusion **51**

References **53**

Appendix A oneAPI Setup Process **59**
A.1 Intel Hardware . . . 59
A.2 FPGA . . . 60
A.3 Nvidia GPUs . . . 61

Chapter 1
Introduction

Radar systems have been widely used in various applications such as defense, air traffic control, and weather monitoring for several decades. However, the development of new technologies such as active electronically scanned array (AESA) radar systems [4] is enabling greater flexibility, higher resolution, and improved performance in cluttered environments. AESA radar systems typically use multi-channel antenna arrays to transmit and receive radar signals, and the received signals are processed to extract useful information such as range, velocity, and direction of targets.

The signal processing requirements for AESA radar systems are particularly demanding due to the need to process data from hundreds or thousands of channels in real-time and the complexity of the algorithms required to extract useful information from the received signals. This requires high-performance hardware, and in addition to traditional central processing units (CPUs), it can be beneficial to use more specialized hardware, such as graphics processing units (GPUs) and field programmable gate arrays (FPGAs). However, CPUs, GPUs and FPGAs traditionally all use different programming models and languages, which makes it difficult to switch out hardware without rewriting large parts of the code.

Advanced radar systems are often produced in low volumes [1], which makes the development cost per unit high and many radar systems are in use for several decades, requiring maintenance as hardware is discontinued. Additionally, it is advantageous to be able increase the performance of the system over its lifespan through more advanced algorithms and upgrading hardware as faster processors become available [2].

Developing radar systems in a cost-effective manner requires focus engineering efficiency and sustainability [2]. Engineering efficiency in the context of software engineering pertains to the complexity of writing code. For example, changing a few parameters of an algorithm is likely less complex than rewriting the whole algorithm in a different programming language. Thus, being able to reuse more of the code increases engineering efficiency. Sustainability is tightly connected to engineering efficiency. A sustainable system should allow for both software upgrades and switching out the hardware, either because the previous hardware was discontinued or to improve cost or size. It should also be possible to scale the system

depending on the context the system is used, for example scale it down for a light drone implementation or up for a powerful ground-based system. Finally, the application design should be independent of the hardware design or have a clear path to replacement in case of the hardware becoming obsolete. Unless it can be guaranteed that the hardware provider will last for the full life cycle of the product, vendor lock-in should be avoided.

To address these challenges, Saab's radar department is investigating the potential of using a generic parallel computing application programming interface (API) called oneAPI [16]. oneAPI is an open programming model created by Intel that includes Data Parallel C++ (DPC++) [35], which implements SYCL [36], a cross-platform abstraction layer that enables code reuse across CPUs, GPUs, and FPGAs. Intel also provides various libraries that integrate well with oneAPI along with profiling and debugging tools [8].

1.1 Purpose

The goal of this Master's thesis is to evaluate to what extent oneAPI can help model high-performance radar signal processing applications in a unified way, without considering specific target processor architectures and thus improve engineering efficiency and sustainability.

1.2 Research Questions

The primary research questions we aim to address are:

- **Productivity**: How much code is needed to setup and implement algorithms with oneAPI and how does it compare to other similar APIs?
- **Portability**: How much of the oneAPI code can be reused for different hardware architectures and what changes are needed?
- **Performance**: How does the computational performance of oneAPI compare to other established libraries?

1.3 Scope

While the research questions are applicable to a broad range of high-performance computing fields, the scope of this thesis is focused primarily on the domain of radar signal processing. The intent is to evaluate the suitability of oneAPI for the unique challenges posed by this area, such as real-time processing of large volumes of data and the long-term maintainability of the software in the face of hardware upgrades and replacements. The choice of radar signal processing also provides a practical and real-world context for our evaluations and analyses.

1.4 Overview

Here is an overview of the following chapters in the thesis:

- **Chapter 2** lays out the foundational concepts of radar systems and the signal processing pipeline used as a benchmark in the thesis. It also describes the how signal processing is well-suited for parallelization on CPUs, GPU and FPGAs.
- **Chapter 3** introduces oneAPI and its different components along with related programming models that are used as comparisons in the thesis.
- **Chapter 4** reviews recent studies on oneAPI, including code conversion and its performance impact, optimization strategies and comparing oneAPI to other programming models.
- **Chapter 5** discusses how experiments are conducted and how they can answer the research questions.
- **Chapter 6** goes through an implementation of an operation in oneAPI and other programming models in order to compare productivity.
- **Chapter 7** presents the results of experiments that evaluate performance of oneAPI.
- **Chapter 8** discusses the results in terms of productivity, portability and performance.
- **Chapter 9** concludes the thesis.

Chapter 2

Radar Signal Processing

This chapter goes over the radar signal processing operations that are used in the thesis.

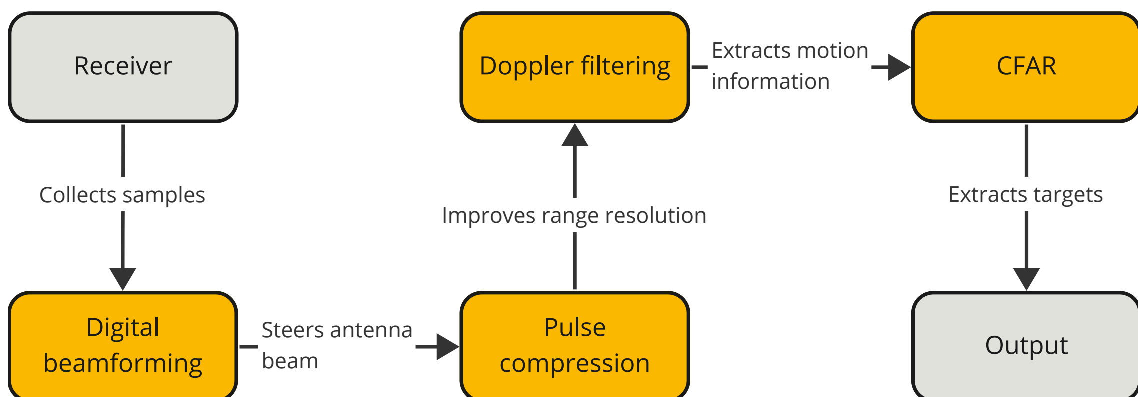

Figure 2.1: Signal processing pipeline used in this thesis.

2.1 Radar Introduction

In radar systems, a transmitter sends out electromagnetic waves or pulses towards a target, which reflects these pulses back to the radar receiver [37]. The receiver then measures the time delay between the transmitted and received pulses, as well as the amplitude and phase of the received pulses.

In an AESA-based multi-channel radar system, each channel has its own transmitter and receiver, which allows for simultaneous measurement of multiple targets at different angles. The pulse echoes received by each channel are typically slightly different in phase, which allows the receiver to distinguish between signals from different directions.

To obtain range information about the target, the radar system samples the received pulses at different times, corresponding to different ranges. The time delay between the

transmitted and received pulses is proportional to the distance to the target. By collecting the sampled signals for multiple ranges, pulses and channels, the radar system can create a *cube* of data. The data cube is a three-dimensional matrix that stores the radar data as complex numbers. The first dimension represents the number of channels in the radar system. The second dimension represents the pulse repetition interval, which is the time between the transmission of two consecutive pulses. Finally, the third dimension represents the range bins, also refered to as range gates, which correspond to the distances of the objects from the radar.

2.2 Signal Processing Pipeline

In order to assess both performance and coding efficiency, we implement a signal processing pipeline, following the methodology presented in an MIT paper [30]. The pipeline operates on data cubes that contain the radar samples.

Samples are collected by the radar receiver before the signal processing is applied. The signal processing includes four primary operations: digital beamforming (DBF) steers antenna beams, pulse compression (PC) improves range resolution, Doppler filtering (DF) extracts motion information, and Constant False Alarm Rate (CFAR) detection extracts targets [37]. Corner turns (CT) are used between operations to improve performance. The signal processing pipeline can be seen in Figure 2.1. The following description details how each operation is realized.

Denote $X \in \mathbb{C}^{N_C \times N_P \times N_R}$ as a data cube that comprises N_C channels, N_P pulses, and N_R range gates. The individual elements in this data cube are denoted by $X[c, p, r]$, representing a specific channel c, pulse p, and range gate r. Moreover, the set of all range gates for a particular channel and pulse is represented by the vector $X[c, p, :]$.

Digital Beamforming: The realization of the digital beamforming operation can be described as a batched matrix multiplication with precomputed weights. If $W \in \mathbb{C}^{N_C \times N_B}$ represents the beamforming weights, where N_B denotes the number of beams, the operation can be expressed as $Y[:, p, :] = W^H X[:, p, :]$. Here, $Y \in \mathbb{C}^{N_B \times N_P \times N_R}$ denotes the output. Within the code, this operation is realized with a general matrix multiplication function, batched over the pulse dimension.

Pulse Compression: Pulse compression can be seen as a convolution with a relatively large filter, applied across range gates. Denoting the filter by $h \in \mathbb{C}^{N_F}$, the pulse compression operation can be represented as $Y[c, p, :] = X[c, p, :] * h$, where $*$ represents the convolution operator. For a larger N_F, efficiency can be improved by performing the convolution in the frequency domain. This involves applying a discrete Fourier transform (DFT) to the input, conducting an elementwise multiplication (EWM) with the filter, and finally applying an inverse DFT to the result. The DFT operations are realized with the Fast Fourier Transform (FFT) algorithm. However, libraries like FFTW [25] require creating an FFT plan prior to executing the FFTs. Generating the plan might test different algorithms to find the most efficient one.

Doppler Filtering: The Doppler filtering stage consists of applying a DFT to the data cube, which is again accomplished by calling an FFT function. In mathematical notation it looks like $Y[c, :, r] = DFT(X[c, :, r])$.

CFAR Detection: The CFAR detection process involves calculating the mean absolute

sum of the values surrounding each element in the range dimension. This technique has two adjustable parameters: the number of values on each side of the tested element that are included in the summation N_{cfar}, and the number of guard gates G, which are the values immediately adjacent to the cell that are omitted from the sum. Mathematically, it looks like $Y[c,p,r] = \frac{1}{2N_{cfar}} \sum_{i=G+1}^{G+N_{cfar}} |C[c,p,r+i]|^2 + |C[c,p,r-i]|^2$.

Lastly, a **corner turn** (CT) operation can be carried out to rearrange the data for better cache locality, thereby optimizing the efficiency of certain operations. Mathematically, a corner turn that swaps the last two dimensions can be described as $Y = CT(X)$, where $Y \in \mathbb{C}^{N_C \times N_R \times N_P}$ and $X \in \mathbb{C}^{N_C \times N_P \times N_R}$. Within the code, the corner turn operation is implemented as a data transfer between two cubes, swapping the indices in the destination during the process.

2.3 Rader Signal Processing Realization

Efficient signal processing realizations are essential to be able handle the massive amount of data produced by AESA radar systems. The linear algebra used in radar signal processing is well-suited for parallelization and can therefore take advantage of hardware accelerators such as GPUs, FPGAs and CPU features like AVX.

2.3.1 Parallelization

Linear algebra is a fundamental mathematical tool in radar signal processing that is used extensively in a wide range of operations, including filtering, compression, and beamforming. These operations can be represented mathematically as linear transformations that operate on vectors and matrices of signal data. In practice, these vectors and matrices can be quite large, making them computationally demanding to process. For example, in our radar signal processing pipeline, the input data cube has dimensions $48 \times 128 \times 3500$ [30], resulting in 21.5 million data samples. Two common and computationally intensive operations are matrix multiplication [40] and Fast Fourier Transform (FFT) [5].

Matrix multiplication can be easily parallelized because each element in the resulting matrix is the dot product of a row from the first matrix and a column from the second matrix. These dot products are independent of each other and can therefore be calculated simultaneously.

In a simple 2D decomposition, the matrices are divided into blocks or tiles, and each processor is assigned the task of computing the result for a specific block. This approach reduces communication overhead between processors and is well-suited for distributed memory systems.

FFT is a key operation in signal processing used for converting a signal from time domain to frequency domain, a fundamental step in radar signal processing.

The Cooley-Tukey radix-2 decimation in time algorithm [5] is one of the simplest FFT algorithms. It breaks down a discrete Fourier transform (DFT) of size N into two interleaved DFTs of size $N/2$ at each stage, and this process can be continued recursively. Each stage of the computation can be performed independently, making it suitable for parallel processing.

Parallelization can be achieved through a variety of techniques, including SIMD, multithreading, and distributed computing. Parallelizing linear algebra operations can signifi-

cantly reduce the processing time required to analyze large volumes of radar data by utilizing more of the available resources of modern processors.

2.3.2 Accelerators

GPUs and FPGAs are well-suited for massively parallel workloads, such as linear algebra, due to their unique architectural characteristics and hardware capabilities.

GPUs: Graphics Processing Units (GPUs) were originally designed for rendering graphics and performing calculations related to computer graphics [43]. However, their massively parallel architecture has proven to be highly effective for general-purpose computation, particularly for linear algebra workloads. Modern GPUs consist of thousands of small, energy-efficient cores that can perform simple arithmetic operations concurrently. For example, the Nvidia Geforce RTX 2070 used in this thesis has 2304 general compute cores. These cores are organized into multiprocessors, which provide a hierarchical structure for managing thread execution and memory access.

Some of the key features that make GPUs well-suited for linear algebra workloads [43] include:

- **High arithmetic throughput**: GPUs can execute a large number of arithmetic operations per second, enabling them to process large-scale linear algebra problems quickly.
- **High memory bandwidth**: GPUs have a large memory bandwidth, which allows them to efficiently access and manipulate large data sets, such as matrices and vectors.
- **Efficient SIMD execution**: GPUs are designed to exploit SIMD (Single Instruction, Multiple Data) parallelism, which enables them to perform the same operation on multiple data elements simultaneously. This is particularly beneficial for linear algebra workloads, as many operations can be vectorized and executed in parallel.

FPGAs: Field-Programmable Gate Arrays (FPGAs) are reconfigurable hardware devices that can be programmed to implement custom logic circuits [23]. Unlike GPUs, which have a fixed architecture and instruction set, FPGAs can be tailored to the specific requirements of a given workload, providing a high degree of flexibility and adaptability. This customization allows FPGAs to achieve high performance and energy efficiency for a wide range of applications, including linear algebra.

Some of the key features that make FPGAs well-suited for linear algebra workloads [35] include:

- **Customizable hardware**: FPGAs can be programmed to implement specialized hardware accelerators for linear algebra operations, such as matrix multipliers or systolic arrays. These accelerators can be optimized for specific problem sizes or data types, enabling high performance and energy efficiency.
- **Fine-grained parallelism**: FPGAs can exploit fine-grained parallelism at the bit or word level, allowing them to perform multiple independent operations concurrently. This is particularly useful for linear algebra workloads, which often involve a large number of small, independent calculations.

- **Low-latency communication**: FPGAs provide low-latency communication, enabling efficient data sharing and synchronization for parallel linear algebra operations. This reduces the overhead of communication and synchronization, which can be a bottleneck for large-scale parallel workloads.

2.3.3 CPU

Central Processing Units (CPUs) are the primary computing engines of most computers. While traditionally associated with serial processing, modern CPUs have evolved to include several features that allow for significant parallel computation. Two such features are multi-core processing and SIMD instruction sets such as AVX (Advanced Vector Extensions) [34].

Multi-core Processing: Modern CPUs consist of multiple cores, each of which can execute instructions independently of the others. This allows for thread-level parallelism where different threads of execution run simultaneously on different cores. Many linear algebra operations can be parallelized at the level of outer loops, making multi-core CPUs well-suited for these types of workloads. For instance, different columns or rows of a matrix can be processed concurrently by different cores. Additionally, modern CPUs support multithreading technologies like Hyper-Threading [39] (on Intel CPUs) which can further increase parallelism by allowing each core to execute multiple threads concurrently.

AVX: Advanced Vector Extensions [34] is a SIMD (Single Instruction, Multiple Data) instruction set extension for x86 CPUs introduced by Intel. SIMD involves performing the same operation on multiple data points simultaneously, which is ideal for workloads that involve repetitive operations on large datasets, such as linear algebra.

AVX provides 128-bit, 256-bit (AVX2), and as of AVX-512, 512-bit wide vector registers that allow for the simultaneous processing of several data points. For example, with AVX2, it is possible to perform eight 32-bit floating-point operations concurrently within a single CPU core. This makes AVX particularly well-suited for linear algebra operations like vector and matrix operations, which often involve applying the same operation to each element in a set of data.

Moreover, AVX includes specific instructions designed to accelerate certain mathematical operations common in linear algebra, such as fused multiply-add, which performs a multiplication and addition in a single operation.

SIMD parallelism is a crucial aspect of modern CPU design that significantly accelerates many workloads, including linear algebra. However, it is important to note that using AVX effectively often requires careful attention to details such as data alignment, memory access patterns, and the structure of the computation to ensure that operations are efficiently vectorized and that the AVX units are fully utilized [29].

Chapter 3

Programming Models

This chapter introduces oneAPI along with other established and similar programming models that are used as comparison.

3.1 oneAPI

Intel's oneAPI [16] builds upon an industry standard called SYCL [26], a unified programming model that enables developers to write parallelized applications for a wide range of heterogeneous systems. An overview of the oneAPI ecosystem can be seen in Figure 3.1. It shows that oneAPI consists of tools, DPC++ [35] and libraries. Intel Advisor and Intel VTune [6, 14] are two examples of the included tools, oneDNN, oneMKL and oneDPL [10, 13, 11] are notable oneAPI libraries and HIP, CUDA, OpenCL and Level Zero [22, 18, 27, 12] are examples of SYCL backends that the DPC++ code can be compiled for.

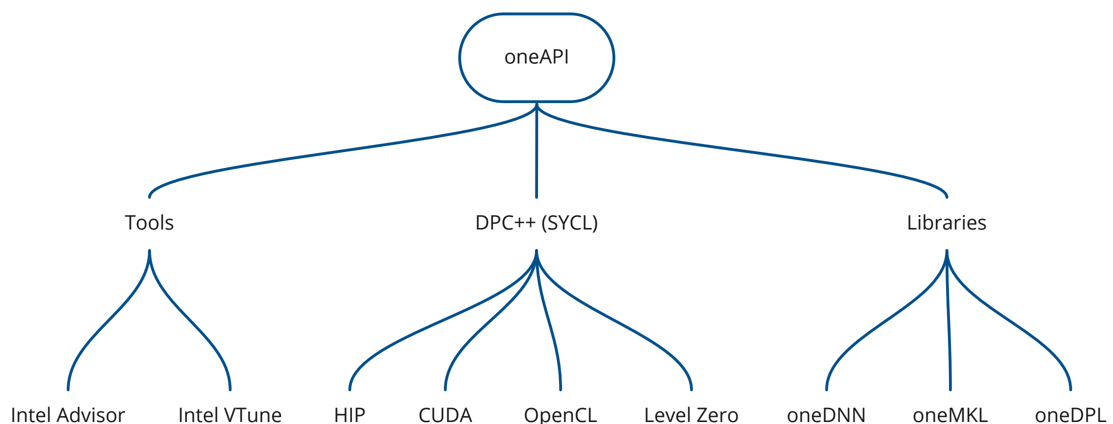

Figure 3.1: Overview of the oneAPI ecosystem.

3.1.1 SYCL

SYCL is a programming model for heterogeneous systems that allows developers to write code in a single-source style using standard C++ [36]. It is based on the OpenCL standard and was designed to enable programming for a wide range of heterogeneous devices such as CPUs, GPUs and FPGAs [26]. SYCL supports modern C++ features such as templates, lambdas, and range-based for loops.

In SYCL, the host and device code are written in the same language, in constrast to OpenCL where the kernels are written in a separate file [27]. The SYCL runtime system is responsible for managing platforms, devices, contexts and memory handling. The SYCL runtime system uses the OpenCL API or other backends like CUDA, HIP and Level Zero to interact with the hardware. SYCL code can be compiled for a generic architecture, such as x86_64, or for a specific device, for example for an Intel Stratix 10 SX FPGA.

One of the key features of SYCL is its ability to express parallelism and data dependencies in a high-level and platform-agnostic manner. SYCL code is written in terms of kernels, which are functions that can be instantiated many times and applied to different input data. Kernels are executed on a device, and their execution is coordinated by the SYCL runtime system. SYCL provides a set of memory and synchronization primitives that allow developers to express dependencies between kernels and data transfers.

SYCL organizes the work to be executed on a device into work groups, subgroups, and work items, which enables developers to control parallelism and data locality. Work items are the smallest units of execution and perform the actual computation. They are grouped into work groups, which are scheduled for execution on the same compute unit, allowing them to share local memory and synchronize execution. Subgroups are an optional division within work groups, providing a finer level of control over synchronization and data sharing.

In SYCL, computation is scheduled with queues. A queue is an object that represents a sequence of command groups that are submitted to a device for execution. Command groups can include data transfers between the host and the device, as well as the execution of kernels. Queues provide a way for the programmer to control the order in which commands are executed and to manage data dependencies between commands.

3.1.2 DPC++

One key part of oneAPI is Data Parallel C++ (DPC++). DPC++ is Intel's implementation of SYCL with extensions, including FPGA-specific functionality and a Unified Shared Memory (USM) model that has now been accepted into the SYCL standard [26]. USM lets you access memory on both host and device through a data pointer. We use USM for all device memory in our signal processing pipeline. DPC++ is supported by Intel's own LLVM-based C++ compiler, *icpx* [9].

3.1.3 Libraries

In addition to DPC++, oneAPI provides a set of libraries with specialized code paths optimized for each supported architecture [16]. These libraries, include functions for linear algebra, signal processing, and machine learning, among others. One of the libraries, oneAPI Math Kernel Library (oneMKL) [13], includes a wide range of numerical algorithms, including

linear algebra functions, FFTs, sparse matrix solvers, and random number generators. Besides the closed source optimized implementations for x86 CPUs and Intel GPUs, there are also open source interfaces that supports multiple backends and devices [15]. Currently, there is partial support for Nvidia and AMD GPUs through their native backends. In addition, the SYCL-BLAS [3] backend provides BLAS functionality implemented in SYCL.

3.1.4 Tools

oneAPI also provides a set of tools for profiling and optimizing applications, including Intel VTune Profiler [14] and Intel Advisor [6]. These tools can be used to analyze application performance, identify bottlenecks, and optimize code for better performance on heterogeneous systems. However, they only support Intel hardware.

3.1.5 DevCloud

The Intel DevCloud service makes oneAPI and Intel hardware more accessible by giving free access to various hardware configurations in the cloud. It is a development sandbox for learning about oneAPI and other cross-architecture APIs. Currently, it features a few different CPUs with integrated GPU and two different FPGAs. We use it to compare oneAPI performance on Intel hardware.

3.2 Related Programming Models

There are several of other programming models similar to oneAPI and they all have their own strengths and weaknesses. The following sections describe the programming models that we compare oneAPI to.

3.2.1 Kokkos

Kokkos is another C++ library and programming model for writing portable parallel applications targeting heterogeneous architectures [24, 41]. Developed by Sandia National Laboratories, Kokkos is designed similarly to SYCL, to abstract the complexities of parallel programming and enable developers to write code that can run efficiently on various hardware platforms, including CPUs, GPUs, and other accelerators.

The Kokkos programming model is based on the concept of parallel patterns, which are high-level constructs that describe the parallelism and data access patterns in an application. Some of the key parallel patterns provided by Kokkos include parallel_for, parallel_reduce, and parallel_scan, which are used to express parallel loops, reductions, and scans, respectively. The parallel_for abstractions in Kokkos and SYCL are very similar, as we will see in Chapter 6.

In Kokkos, data structures are organized using multidimensional arrays, called `View`, which provide a flexible and portable way to manage memory [33]. `View` can be used to represent data in various memory spaces, such as host memory or device memory, and can be resized and reallocated as needed. Additionally, Kokkos provides memory management facilities to automatically handle data transfers between different memory spaces, switch between row-major and column-major memory layout and ensure proper data synchronization.

Kokkos also provides an execution space abstraction, which represents the different hardware resources available for parallel execution and is analogous to SYCL backends. Some examples of execution spaces include C++ threads, OpenMP, and CUDA. It also has an experimental SYCL backend, that is not considered stable yet. By specifying the execution space when defining parallel patterns, developers can control where the computation is performed.

We choose Kokkos as a comparison because of how similar it is to oneAPI, while being more established. We consider Kokkos to be a direct competitor to oneAPI.

3.2.2 Eigen

Eigen is a high-performance C++ template library for linear algebra, matrix and vector operations, and geometrical transformations [28]. Developed as an open-source project, Eigen is designed to provide a flexible and efficient library for a wide range of applications, including scientific computing, computer graphics, robotics, and machine learning.

One of the key features of Eigen is its expressive syntax, which more closely resembles mathematical notation than a Basic Linear Algebra Subprogram (BLAS) API like oneMKL. This syntax, accomplished through C++ operator overloading, allows developers to write complex linear algebra expressions in a concise and readable manner.

Eigen's performance is achieved through a combination of template metaprogramming and vectorization. Template metaprogramming enables Eigen to generate efficient code at compile time by unrolling loops and optimizing expressions based on the matrix and vector dimensions. Vectorization, on the other hand, leverages the SIMD capabilities of modern processors to perform multiple operations in parallel.

For discrete Fourier transforms, Eigen supports three different backends. One of them is FFTW [25], which is an open source project by researchers at MIT that includes high performance FFT implementations.

We use Eigen with the FFTW backend as a baseline because of its high CPU performance, popularity and usage in the industry.

3.2.3 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface developed by Nvidia [18]. It provides a C++-based programming model that enables developers to efficiently utilize Nvidia GPUs' massive parallel processing capabilities for general-purpose computing.

CUDA organizes computation into a hierarchy of threads, blocks, and grids. Threads are the smallest unit of execution and can be grouped into blocks. A block is a collection of threads that can communicate and synchronize with each other. A grid is an array of blocks, and multiple grids can be launched concurrently. This hierarchy allows developers to express parallelism at different granularities, enabling the efficient use of GPU resources for a wide range of computational tasks.

One of the key advantages of CUDA is its extensive ecosystem, which includes a variety of libraries, tools, and resources that simplify the development of GPU-accelerated applications. Two important libraries are cuBLAS [17] and cuFFT [19], which provide optimized implementations of commonly used algorithms in linear algebra and signal processing, respectively.

In comparison to SYCL and oneAPI, CUDA is a platform specific solution, which means that it is tied to Nvidia GPUs and does not support other types of devices, such as CPUs, FPGAs, or other vendor GPUs. However, CUDA is widely adopted [32] and has a mature ecosystem, making it a popular choice for developing high-performance applications that target Nvidia GPUs.

We compare oneAPI with CUDA, since it is the dominant programming model for GPU compute and to see how well oneAPI compares with native code on third-party hardware.

Chapter 4
Related Work

Several recent works have evaluated oneAPI with different approaches. Many of them migrate code written in CUDA to DPC++ using a conversion tool, DPC++ Compatibility Tool (dpct), but also from C++ code.

Costanzo et al. [21] (2021) discuss their experiences porting two small CUDA applications to DPC++ using the dpct conversion tool, a matrix multiplication and a reduction. They include performance numbers for the matrix multiplication running on an Nvidia Geforce RTX 2070 GPU, Intel Core i9-10920X CPU and an Intel Iris Xe MAX GPU. The RTX 2070 is tested both running the original CUDA code and the converted DPC++ with an 8-38% slower result with DPC++. Furthermore, DPC++ on the RTX 2070 runs 7-8 times faster than the CPU and the Xe Max GPU runs 1.5-2.3 times faster than the CPU. Our thesis extends these tests with more operations and investigates and further comparisons with similar programming models.

Wang et al. [44] (2021) migrate a CUDA-based medical ultrasound application to DPC++ and run it on Intel CPU, GPU and FPGA. The application uses a processing pipeline that includes beamforming, envelope detection, log-compression and scan-conversion. Wang et al. find that code conversion is efficient using the dpct conversion tool. To improve performance on GPUs, they use a low-level extension for explicit SIMD instructions to increase vectorization. For FPGA, a pipelining approach with local buffers is used and they note that their optimizations require professional FPGA experience. When comparing the performance of an Intel Core i7-8700K GPU with its integrated Intel UHD Graphics 630 GPU, the GPU is 9.7 times faster, even though it only has 24% more single precision floating point operations per second. However, the paper does not mention any CPU optimizations and given that the code was originally written for Nvidia GPUs it might not be well-suited for the CPU. The Intel Arria 10 FPGA achieves around double the through put of the Intel UHD Graphics 630. In constrast to [44], our thesis does not base our test application on already optimized GPU code and our results are more focused on performance with as few architecture specific optimizations as possible.

Marinelli et al. [31] (2021) port a highly-optimized GPU-based hash join algorithm to

DPC++ from CUDA using dpct. They execute the code on an Intel CPU and two GPUs and an Nvidia GPU and compare with hand-optimized CUDA. An Intel Gen 9 integrated GPU is around 40% slower than its Intel Xeon E-2176G CPU, while an Intel Xe Max GPU is around twice as fast as the CPU. DPC++ on the Nvidia Geforce RTX 2080ti is around 10 times faster than the CPU. When comparing the DPC++ implementation to the original CUDA code, the DPC++ version is 1.3-4.7 times slower on the RTX 2080ti. Marinelli et al. also compares the performance of using different work group sizes and show that it can have a significant impact. The paper does not mention trying automatic work group size. In our thesis, we explore whether automatic work group size can eliminate the manual tuning.

Volokitin et al. [42] adapts a particle simulation implemented in C++ to use DPC++ and run it on an Intel CPU and two GPUs. They find porting the C++ code to DPC++ to be easy and that the DPC++ performance is only around 10% worse than the C++ code using OpenMP on the CPU. In constrast to the other papers mentioned, the code in this paper is based on, and compared with, an optimized CPU implementation. We further compare CPU implementations with Eigen, FFTW and Kokkos.

Besides extending the performance evaluations done in the mentioned papers, we also compare productivity of oneAPI with similar programming models.

Chapter 5
Approach

To evaluate the productivity, portability and performance of oneAPI, we conduct a series of experiments. The experiments compare various aspects of oneAPI to other programming models, namely Eigen with FFTW, Kokkos and CUDA. We deemed these programming models to be representative of each aspect we wanted to evaluate. Eigen and FFTW are well established math libraries for CPUs and therefore suitable as a baseline. Kokkos is similar to oneAPI in terms of its high-performance computing and portability goals and has also been around for a longer time. Finally, CUDA is the most frequently mentioned parallel programming language in research according to Google Scholar [32] and the native language for programming Nvidia GPUs and therefore a suitable comparison for evaluating oneAPI's performance on third-party hardware.

5.1 Productivity

The productivity aspect is evaluated on the basis of how much code and parameter tuning is needed to achieve the desired functionality. We evaluate it on the code we write for the performance experiments and thus compare oneAPI to the other programming models used. More specifically, we look at what setup is needed before you can launch a kernel, what parameters should be tuned and how much extra code is required for the kernels besides the core algorithm. This is discussed in Chapter 6.

5.2 Portability

Since the main selling point of oneAPI is that it improves portability and avoids vendor lock-in, we run the code on several hardware architectures and observe if it compiles and runs successfully. We judge the portability based on how much of the code needs to be changed to make it run and further changes to make it run well. Portability is also evaluated on the code

written for the performance experiments and the results are partly presented when describing the performance experiments in the coming sections, but also summarized in Chapter 8.

5.3 Performance

The performance of oneAPI is evaluated in several ways. We compare performance to a baseline, investigate the trade-off between automatic or manual tuning of work group size, sensitivity to input size with non-optimal factors, scaling across different architectures and input sizes and compare kernel performance to a competitor and native code.

All benchmarks are run for 10 iterations to achieve more consistent timings. Before measuring, a warmup sequence is run, that includes all the timed operations. This ensures that any initialization and JIT compilation is completed before the benchmark and not included in the measurements. Since the timings still varies a few percent between runs, the benchmarks are run multiple times for each configuration and the best time is recorded. Five times was deemed enough to consistently get at least one good result and the best time was chosen, because it is likely the least affected by background processes and other random occurrences that can harm performance. All code is compiled with only the `-O3` flag, using Intel's *ipcx* compiler for the CPUs and Intel GPU and Intel's *clang* compiler for the Nvidia GPU with *nvcc* [20] for the CUDA kernels.

The results are presented in Chapter 7. Next, we describe each experiment in more detail.

5.3.1 Comparison with Baseline

First, we want to make sure that it is competitive compared to established math libraries that we consider as baseline. If other libraries offer higher performance, then the improved portability of oneAPI might not be enough for it to get chosen, although the exact performance threshold depends on the context. To verify that oneAPI is competitive with other popular math libraries, we compare its performance with the open source library Eigen, with the FFTW backend for FFTs. With Eigen, multithreading is enabled by OpenMP.

Pulse compression is used as the benchmark which consists of applying an FFT on the input, elementwise multiplication (EWM) with a filter and an inverse FFT. Pulse compression was chosen because it provides more breadth of algorithms than the other operations in the pipeline. We deem it sufficient to get a sense of whether oneAPI can compete with the Eigen, without having to re-implement the whole pipeline. Instead of running the experiment on an Intel CPU through Intel Devcloud, we run this experiment on our local machine with an AMD Ryzen 5 3600 CPU, since it makes it easier to install additional libraries such as Eigen.

5.3.2 Importance of Tuning Work Group Size

Next, one of the architecture specific parameters SYCL exposes to the programmer is work group size. We want to evaluate how much impact it has on performance and how well the implementation can automatically tune it. The optimal work group size depends on the device, so it would make oneAPI less portable if you have to tune it for each device to achieve good performance.

In SYCL, the work group size determines how many work items, instances of the kernel, are scheduled to the same compute unit. What exactly a compute unit represents in hardware is architecture dependent, but for CPUs and GPUs, setting work group size to one masks out all SIMD units other than the first one [35]. Intel Gen 12 GPUs, for example, can execute eight 32-bit operations in parallel per instruction [7], so only utilizing one of those should be detrimental to achieving good performance. We run the experiment on an Intel Gen 12 UHD Graphics via Intel Devcloud. We choose a GPU for this experiment, because we expect it to be more affected by the choice of work group size than a CPU, since GPUs have weaker individual cores than CPUs and are more reliant on utilizing many of them.

To determine how important it is to choose proper work group sizes, we use three different work group size configurations. For the first configuration we set work group size to one, to represent the worst case. Then, SYCL lets you omit specifying work group size, in which case the SYCL implementation decides for you, which is the second configuration. For the last configuration, we manually tune the work group size by measuring the performance of different configurations. We notice that the performance of the automatic tuning varies significantly depending on input size. Therefore, we include two different input sizes, one with range dimension 3500 and one with 3584. In this experiment we only include the CFAR and corner turn, since those are the only operations we implemented as SYCL kernels and specify work group size for.

5.3.3 Intel CPU vs Integrated GPU

We notice that the oneMKL FFT performance is more sensitive to input size on GPU than CPU. Therefore, we compare the performance gains of padding the input on CPU and GPU. Having to pad input for certain devices, would make oneAPI less portable and add another device specific parameter.

We compare the performance on the Intel i9 11900KB CPU with its integrated UHD Graphics GPU running in Intel Devcloud. We also tested padding the range dimension of the input data from 3500 to 3584 and 4096 for better FFT performance. This experiment includes the full signal processing pipeline. To make each device perform its best, we tuned work group size for both individually and reused FFT plans on GPU. We reused the FFT plans for the GPU, since creating them for the GPU is slow, sometimes slower than the actual FFT. The reason for not reusing FFT plans on the CPU is that it made the inverse FFT three times slower. However, creating the FFT plans for the CPU each time did not significantly increase the total time.

5.3.4 Discrete GPU

To evaluate portability of oneAPI beyond Intel hardware, we add a third-party GPU to the comparison, the Nvidia Geforce RTX 2070 running on our local machine. In this experiment all devices use a range dimension padded to 3584. The only modification required compared to the code running on the UHD Graphics GPU is that for the Nvidia GPU we have to use the native cuFFT library directly instead of through the oneMKL interface, since it has not yet been implemented.

5.3.5 CPU: oneAPI vs Kokkos

The code for kernels in oneAPI and Kokkos is very similar, so one would also expect the performance to be about the same. First we run the test on our local machine with a AMD Ryzen 5 3600 CPU, again because of the difficulties of using Kokkos in Intel Devcloud. We did not get Kokkos' experimental SYCL backend to work for CPU nor the OpenMP backend, so we used the C++ threads backend that use standard library threads for parallelization. We compare the elementwise multiplication used in pulse compression, CFAR and corner turn, since those operations are written as kernels.

5.3.6 GPU: oneAPI vs Kokkos vs CUDA

We repeat the same kernel benchmark with the Nvidia GPU. This time we add kernels written in CUDA to the comparison to see how oneAPI compares to native code on third-party hardware. For the GPU kernel comparison, the Kokkos code uses the SYCL backend.

5.3.7 Scaling Problem Size

The final experiment shows how well oneAPI performs when scaling problem size on different devices. It also gives an indication of whether oneAPI adds significant overhead on the three different devices tested, Intel Core i9 11900KB, Gen 12 UHD Graphics and Nvidia Geforce RTX 2070. The native CUDA kernels are again added to the comparison to see if oneAPI behaves any differently. We run full signal processing pipeline and scale the range dimension from 128 to 16384, doubling it for each step.

5.3.8 FPGA

We compile and run the code on the Intel Stratix 10 SX. However, according to a runtime error the oneMKL functions used are not supported on the device. When running the code without those functions, the program hangs indefinitely. Because of the four hours long compile times, lack of FPGA expertise and time, we did not debug it any further.

5.4 GPU Considerations

Offloading computations onto a discrete GPU (not integrated with the CPU) usually comes with various overheads compared to running everything on the CPU and therefore requires some extra consideration when comparing performance. For example, launching a CUDA kernel on an Nvidia GPU involves setting up the execution context [18], including the grid and block dimensions, shared memory and registers. This overhead can become significant when launching a large number of small kernels. Discrete GPUs have their own memory, separate from the main system memory that CPUs use and transferring large amount of data between can be slow. Furthermore, synchronizing the entire GPU with all of its cores is another time consuming operation. The first time you use CUDA in a program, there is additional overhead to initialize the CUDA context on the GPU. This can be quite significant,

but it is a one-time cost per program execution. CUDA can be compiled to the intermediary programming language PTX [20], analogous to Java bytecode. PTX needs to be just-in-time (JIT)-compiled to binary code at runtime, which results in another initial overhead.

Depending on the use case, many of the mentioned latencies can be amortized or hidden. We assume the real world use case of oneAPI in the context of radar signal processing would be running the signal processing pipeline continuously as more radar samples are made available to the computing device. Therefore, we do not include any initialization or JIT overhead when timing our performance tests. Memory transfers only need to happen at the start and end of the signal processing pipeline. Assuming the compute time is longer, the transfers can be performed asynchronously while computing to hide the latency cost and are therefore also excluded from the measurements. The need for synchronization is highly use case dependent. Due to an issue with the built-in SYCL profiling API, we have to synchronize after each operation to be able to measure their time costs and it is therefore included. Finally, the kernel launch overhead can be reduced by launching multiple kernels concurrently. However, in this case all operations happen sequentially, so it is not applicable.

Chapter 6

Implementation

In this chapter, we discuss the implementation of the signal processing pipeline and compare the kernel implementations in DPC++, Kokkos and CUDA in order to evaluate the productivity of oneAPI as defined in Chapter 5.

6.1 Code Comparison

The signal processing pipeline was fully implemented using DPC++ and oneMKL. More specifically, oneMKL was used for FFTs and matrix multiplication while elementwise multiplication, corner turns and CFAR were implemented in SYCL kernels. To be able to compare the performance and ease of use, parts of the benchmark were also implemented in Kokkos, Eigen and CUDA, as described in Chapter 5. For all frameworks, we use shared USM memory, which means the memory can be accessed from both host and device. The memory is implicitly copied between the host and device when needed. This feature improves productivity by reducing the need to manually manage memory and is available in all three programming models. If performance is an issue, the allocations can later be switched to host or device and be copied explicitly.

To illustrate the differences between DPC++, Kokkos and CUDA, we use a corner turn as a simple example of how kernel is written. The `corner_turn_mon` function swaps the last two dimensions of a data cube out of place, so $Y = CT_{mon}(X)$, where input data cube $X \in \mathbb{C}^{M \times N \times O}$ and output data cube $Y \in \mathbb{C}^{M \times O \times N}$.

In the DPC++ and CUDA examples we use a simple `Cube` wrapper class that wraps the USM data pointer, keeps track of sizes and provides helper methods like `Cube::I` which converts a 3D index to a 1D index. Kokkos, on the other hand, provides its own multidimensional array class, `Kokkos::View`. It has much more capabilities, letting you easily change the memory layout between row-major and column-major, automatically pads the memory for the target device and is supported in kernels.

6.1.1 DPC++

The `corner_turn_mon` function takes a `sycl::queue` and an input and output `Cube` as arguments. First, we extract the USM data pointers from the `Cube` objects that is due to the kernels not currently supporting modifying the data elements through class methods. In future versions of DPC++, the `mdspan` class, introduced in C++23, will likely be the preferred way of handling multidimensional USM data. It will be possible to read and write to `mdspan` in a kernel, simplifying the code compared to our `Cube` data wrapper.

Next, we define the work group size along with the range we want to loop over. Certain devices such as GPUs require uniform work group sizes, so each range dimension has to be divisible by its work group dimension. Then, we submit a command group to the device queue, which consists of a parallel for loop. The parallel for loop takes the range and work group size as arguments along with the actual kernel that implements the corner turn. Inside the kernel, we extract the global indices and copy from the input memory to the output memory. Since we potentially made the range larger than the actual data dimensions to make it evenly divisible by work group size, we have to check that we are not copying out of bounds.

Finally, submitting the command group generates a `sycl::event` that can be used to define data dependencies for which order submitted kernels can be executed and synchronization. However, all of the operations in our signal processing pipeline should happen in order, so we use an in-order `sycl::queue` that does not permit any reordering.

```
1  template <typename T, size_t M, size_t N, size_t O>
2  sycl::event corner_turn_mon(sycl::queue& q, Cube<T,M,N,O>& x_cube,
   ↪  Cube<T,M,O,N>& y_cube) {
3    /*
4    * Because of kernel constraints we cannot use Cube methods to
5    * write to the USM memory, so we extract the USM data pointers.
6    */
7    auto x = x_cube.data();
8    auto y = y_cube.data();
9
10   /*
11   * Some devices require uniform work group sizes, so each range
12   * dimension has to be divisible by its work group dimension.
13   */
14   auto wgs = range(1, 8, 8);
15   int m = (N / wgs.get(0) + 1) * wgs.get(0);
16   int n = (N / wgs.get(1) + 1) * wgs.get(1);
17   int o = (O / wgs.get(2) + 1) * wgs.get(2);
18   auto r = range(m, n, o);
19
20   /*
21   * Here we submit a lambda function containing
22   * a command group to the device queue.
23   */
24   auto event = q.submit([&](handler& h) {
25     /*
```

```
26      * We call parallel_for with a kernel consisting of
27      * another lambda function. The function is scheduled
28      * in parallel and run for every element in the range r.
29      */
30     h.parallel_for(nd_range(r, wgs), [=](nd_item<3> item) {
31       int i = item.get_global_id(0);
32       int j = item.get_global_id(1);
33       int k = item.get_global_id(2);
34       if (i < M && j < N && k < O)
35         y[y_cube.I(i, k, j)] = x[x_cube.I(i, j, k)];
36     });
37   });
38
39   // We return the event generated by the submitted command group.
40   // It can be used to define data dependencies or synchronization.
41   return event;
42 }
```

6.1.2 Kokkos

The Kokkos code is very similar to DPC++, but requires slightly less code. Instead of having to pass around a `sycl::queue`, kernels are submitted to a global context that is assumed to be initialized. The kernel itself simply has the indices as parameters rather than a wrapper object, `nd_item`, which you then have to extract them from. However, `nd_item` does provide more functionality like getting the work group index or a subgroup. With the `Kokkos::View`, we can access and write to elements inside the kernel, instead of having to calculate the 1D indices and access them through the pointer. Work groups sizes are called tiles in Kokkos terminology, but similarly specified when submitting the kernel. Finally, instead of only providing the upper range bounds, we also have to specify the lower bounds, which is the one area in this example where Kokkos is more verbose than DPC++.

```
1  using CubeView = Kokkos::View<complex<float>***, Kokkos::LayoutRight,
    ↪  Kokkos::Experimental::SYCLSharedUSMSpace>;
2
3  void corner_turn_mon(CubeView& x_cube, CubeView& y_cube) {
4    int M = x_cube.extent(0);
5    int N = x_cube.extent(1);
6    int O = x_cube.extent(2);
7
8    int wgs = 8;
9    int m = (M / wgs + 1) * wgs;
10   int n = (N / wgs + 1) * wgs;
11   int o = (O / wgs + 1) * wgs;
12
13   Kokkos::parallel_for(
14     Kokkos::MDRangePolicy<Kokkos::Rank<3>>({0, 0, 0}, {m, n, o}, {wgs,
    ↪  wgs, wgs}),
15     KOKKOS_LAMBDA(int i, int j, int k) {
```

```
      if (i < M && j < N && k < O) {
        y_cube(i, k, j) = x_cube(i, j, k);
      }
    }
  );
}
```

6.1.3 CUDA

The CUDA code differs a bit compared to DPC++ and Kokkos in how the work group sizes are specified. In CUDA each kernel is launched with a grid size and a block size argument using the `<<<`grid size, block size, shared memory size, stream`>>>` syntax. A block is a group of threads that run in parallel and it is the CUDA version of a work group. The grid defines how many of the blocks that are scheduled. This is similar to the loop bounds in DPC++ and Kokkos, but you have to do the math of how many are needed to cover the range yourself. For Nvidia GPUs, shared memory is similar to L1 cache and shared between all threads in the same block. In this example we do not use any shared memory, but we do specify a stream. A CUDA stream is a sequence of operations that execute on the device in the order they are issued by the host, analogous to an in-order `sycl::queue`. To make sure that all the signal processing operations happen in the correct order we run them in the same stream.

Here we have to explicitly implement a device function to calculate the 1D index, instead of using the `Cube::I` function defined in host code. The `__device__` specifier means the function can only be called from the device, likely in a kernel. `__global__` functions can be called from both host and device. In `corner_turn_mon_kernel` we use the special objects `blockIdx`, `blockDim` and `threadIdx` to calculate the indices. Compared to the DPC++ and Kokkos code, this slightly increases code complexity, but also flexibility. Again, similar values can be obtain from the DPC++ `nd_item`. Finally, instead of a calling `parallel_for` we use `cuda_host_task` which itself calls the DPC++ function `host_task` and wraps the CUDA function call with synchronization. The synchronization makes sure that all previously submitted operations in the stream have finished before proceeding. In this case, that is redundant since we only use one stream, but we included it to follow how the open source oneMKL CUDA interface was written.

```
__device__ int get_idx(int i, int j, int k, int n, int o) {
  return i * n * o + j * o + k;
}

__global__ void corner_turn_mon_kernel(std::complex<float>* x,
    std::complex<float>*  y, int m, int n, int o) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int k = blockIdx.z * blockDim.z + threadIdx.z;
  if (i < m && j < n && k < o) {
    int idx = get_idx(i, j, k, n, o);
    int idx_t = get_idx(i, k, j, o, n);
    y[idx_t] = x[idx];
  }
}
```

```
15
16 void corner_turn_mon(std::complex<float>* x, std::complex<float>* y, int
   ↪ m, int n, int o, cudaStream_t stream = 0) {
17   dim3 block(4, 1, 64);
18   dim3 grid((m + block.x - 1) / block.x, (n + block.y - 1) / block.y, (o +
   ↪ block.z - 1) / block.z);
19   corner_turn_mon_kernel<<<grid, block, 0, stream>>>(x, y, m, n, o);
20 }
21
22 template <typename T, size_t M, size_t N, size_t O>
23 void corner_turn_mon(sycl::queue& q, Cube<T,M,N,O>& x_cube, Cube<T,M,O,N>&
   ↪ y_cube) {
24   auto x = x_cube.data();
25   auto y = y_cube.data();
26
27   q.submit([&](handler& cgh) {
28     cuda_host_task(cgh, q, [=](CudaScopedContextHandler &sc) {
29       auto stream = sc.get_stream(q);
30       corner_turn_mon(x, y, M, N, O, stream);
31     });
32   });
33 }
```

6.2 Productivity Results

Writing DPC++ kernels is similar to both Kokkos and CUDA. However, DPC++ has the down side of having to explicitly pass and submit command groups to a `sycl::queue`, while Kokkos and DPC++ lets you implicitly use a default queue or stream, which can reduce code complexity. The three programming models all let you specify range and work group size, but for DPC++ the latter is optional which reduces the need for coming up with your own heuristics or manually tuning it. Finally, Kokkos has a useful data wrapper that simplifies using multidimensional data in kernels, which neither DPC++ will not have until `mdspan` is introduced with C++23.

Chapter 7
Performance Results

In this chapter, we present the results from the performance evaluations described in Chapter 5.

7.1 Comparison with Baseline

Figure 7.1 shows the results of the comparison between oneAPI and Eigen with FFTW. The y-axis shows the time it took to run the pulse compression operation for 10 iterations. The stacked bars show the routines that make up pulse compression, FTT in dark gray, elementwise multiplication in light gray, inverse FFT in yellow and output scaling in beige. The reason for only Eigen having the scale operation is that for oneMKL, output scaling is included in the API for the inverse FFT. However, the Eigen API does not do that, so we do the scaling explicitly after the inverse FFT, which greatly contributes to the total time. The scaling is needed to get the correct output, since setting the scale factor to $1/n$, where n is the number of elements in X, results in $IFFT(FFT(X)) = X$.

Another difference is in the FFT configuration. To make the comparison fair, we use the fastest FFT configuration that we find for each API. In implementation is that FFTW lets you choose how thoroughly to plan the FFT. If you choose one of the slower options like `FFTW_MEASURE` it runs the FFT with different parameters and measures which combination is fastest. We use `FFTW_MEASURE` and reuse the FFT plans for all test iterations, but do not include the planning time in our timings. For oneMKL, there is no such option, but the default strategy is so fast that it is not very significant in relation to running the FFT. Therefore, the oneMKL FFT plans are not reused and the planning is included in the timings. We did also try reusing the oneMKL FFT plans, but it made the inverse FFT around three times slower on the CPU, which we suspect to be a bug.

The result shows that oneAPI is about twice as fast as Eigen for the pulse compression operation. While the explicit scaling contributes a large part of time cost for Eigen, elementwise multiplication is also individually around twice as fast.

7.2 Importance of Tuning Work Group Size

The next experiment investigates how much difference tuning work group size makes and the results can be seen in Figure 7.2. Again, y-axis shows the time for 10 iterations, but this time of CFAR and corner turn kernels, in dark and light gray, respectively. There are three bars: the first shows work group size set to one, the second letting the implementation decide and the third is manually tuned. There are results for two different range dimensions, 3500 and 3584.

As expected, the performance is much better when properly tuned than when set to one, with manually tuned work group size being around 10 times faster. The more interesting result is how well automatic work group size works. For range size 3500, automatic is around 50% slower than the manually tuned result. However, for certain input sizes it fails and seemingly chooses a bad work group size for the corner turn kernel, with performance closer to setting work group size to one than manual tuning. Since the failure only happens for corner turn and a certain input size in this experiment, it might be a DPC++ bug, but nevertheless makes automatic tuning less reliable.

7.3 Intel CPU vs Integrated GPU

In Figure 7.3, the y-axis shows the time for running 10 iterations of the full signal processing pipeline on an Intel Core i9-11900KB and its integrated UHD Graphics GPU. Digital beamforming (DBF) is shown in dark gray, pulse compression (PC) in light gray, Doppler filtering (DF) in yellow, CFAR in beige and corner turn (CT) in blue. The pipeline is run for three different range sizes for each device, 3500, 3584 and 4096.

FFT performance depends heavily on having good size factors, especially on GPUs. Parallel FFT algorithms usually support input sizes that can be written as $2^a \cdot 3^b \cdot 5^c \cdot 7^d$, where $a, b, c, d \in \mathbb{N}$ (sometimes with more prime factors), but they perform best for as small factors and as few different factors as possible. A large prime number should be avoided at all costs and simple way to accomplish that is by zero padding the input. While $3500 = 2^2 \cdot 5^3 \cdot 7$ are not the worst factors, we can see in Figure 7.3 that on the integrated GPU, $3584 = 2^9 \cdot 7$ is significantly faster and even $4096 = 2^{12}$ is a bit faster even though it is 17% larger. However, having good prime factors does not have as large impact on the CPU performance.

The results show that the integrated GPU with proper padding is overall significantly faster than the CPU for the full signal processing pipeline. The matrix multiplication in digital beamforming and FFTs in pulse compression and doppler filtering perform similarly on both processors. Transpose and CFAR, on the other hand, is where the integrated GPU pulls ahead.

7.4 Discrete GPU

In Figure 7.4, we add the discrete Nvidia Geforce RTX 2070 GPU to the comparison with the Intel Core i9-11900KB CPU and UHD Graphics GPU. The setup is the same as in Section 7.3, except the input has range dimension 3584 for all devices. The RTX 2070 GPU is around 10 times faster than the CPU.

7.5 CPU: oneAPI vs Kokkos

For the oneAPI and Kokkos kernel comparison we compare three kernels and in Figure 7.5, elementwise multiplication (EWM) can be seen in dark gray, CFAR in light gray and corner turn (CT) in yellow. The y-axis shows the time for running the kernels for 10 iterations on the an AMD Ryzen 5 3600. oneAPI is slightly faster for all the kernels. The elementwise multiplication kernel stands out, being around 67% faster in oneAPI than Kokkos.

7.6 GPU: oneAPI vs Kokkos vs CUDA

The results in Figure 7.6 are presented the same way as in Section 7.5, but run on the Nvidia Geforce RTX 2070 and kernels written in CUDA is added to the comparison. oneAPI outperforms Kokkos across all kernels, but the CUDA kernels are even faster. The elementwise multiplication kernel in CUDA is about ten times faster than the Kokkos version, while CFAR and corner turns are around twice as fast.

7.7 Scaling Problem Size

In this experiment we run the full signal processing pipeline on the Intel Core i9-11900KB, Intel UHD Graphics and Nvidia Geforce RTX 2070. In addition, to using oneAPI for all devices, we also include the RTX 2070 running the CUDA kernels for elementwise multiplication in pulse compression, CFAR and corner turns. The range dimension is scaled from 128 to 16384, doubling it for each step and the signal processing pipeline is run for 10 iterations for each range size and device. The timings are then divided by the range dimension, so the resulting metric shows how well the device performs per range bin. We further normalize so that the time shown for range size 2048 is equal to one.

The results are shown in Figure 7.7. It can be seen that both the CPU and the UHD Graphics GPU perform most efficiently at small problem sizes, while the discrete GPU excels at large sizes. Again, we can see that the CUDA kernels scale better to large problem sizes than the oneAPI versions. For oneAPI, the performance starts degrading at range dimension 8196, while the CUDA version stays the same.

To investigate the impact of synchronization on the Nvidia GPU, we measured the CUDA code again with range 128, but without the timing synchronization. Then we saw a performance gain of around 50%.

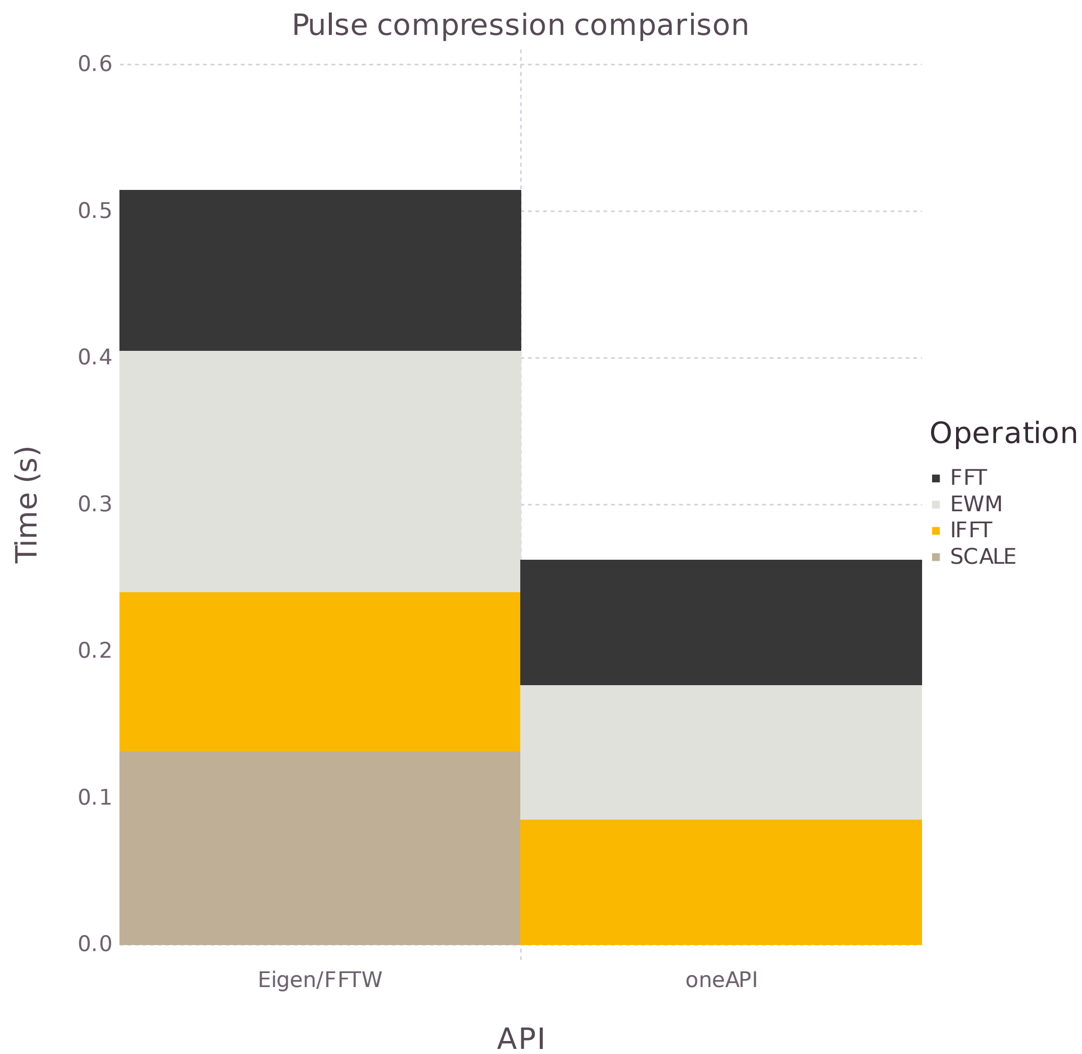

Figure 7.1: Timings for 10 iterations of pulse compression using Eigen/FFTW and oneAPI.

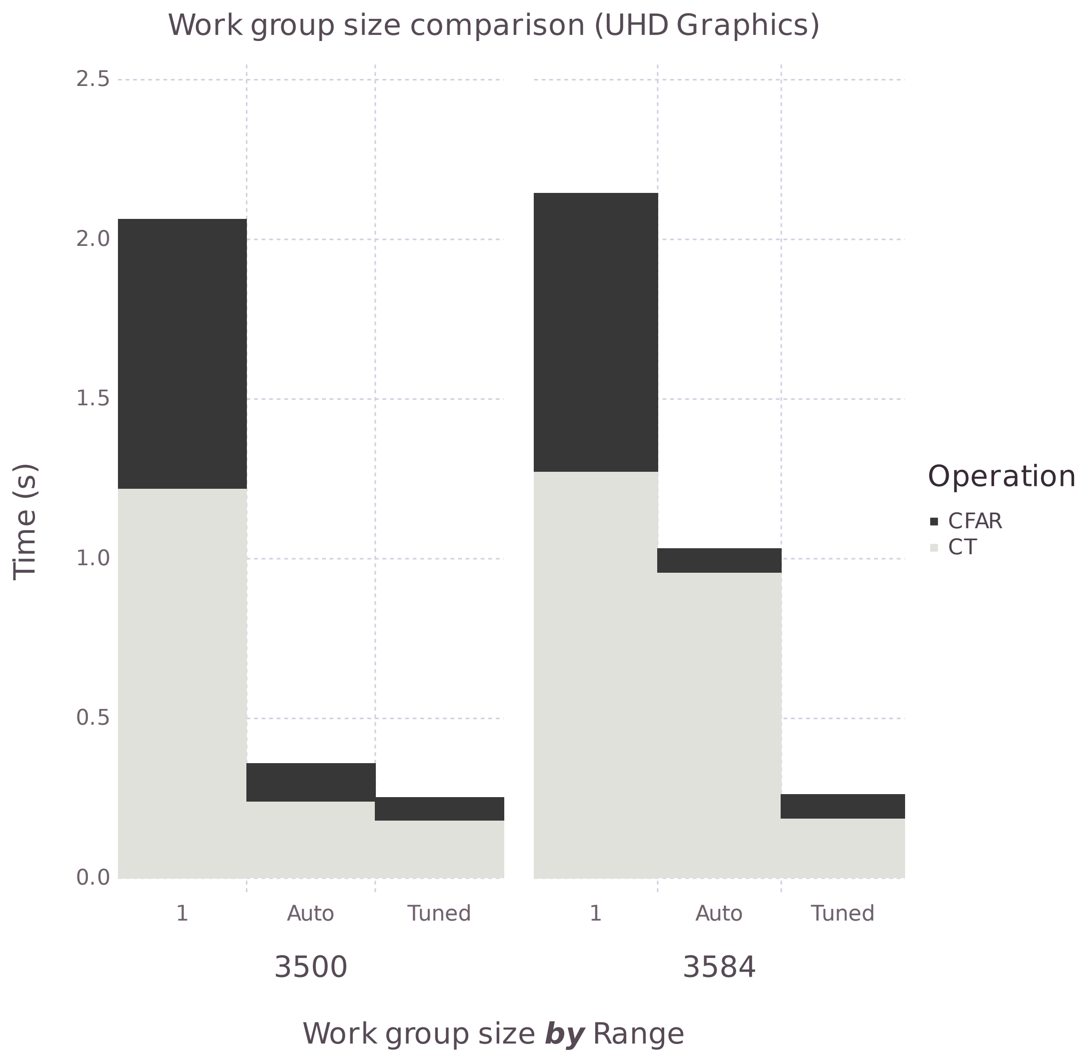

Figure 7.2: Timings for 10 iterations of the full pipeline with different work group sizes and range dimension padding.

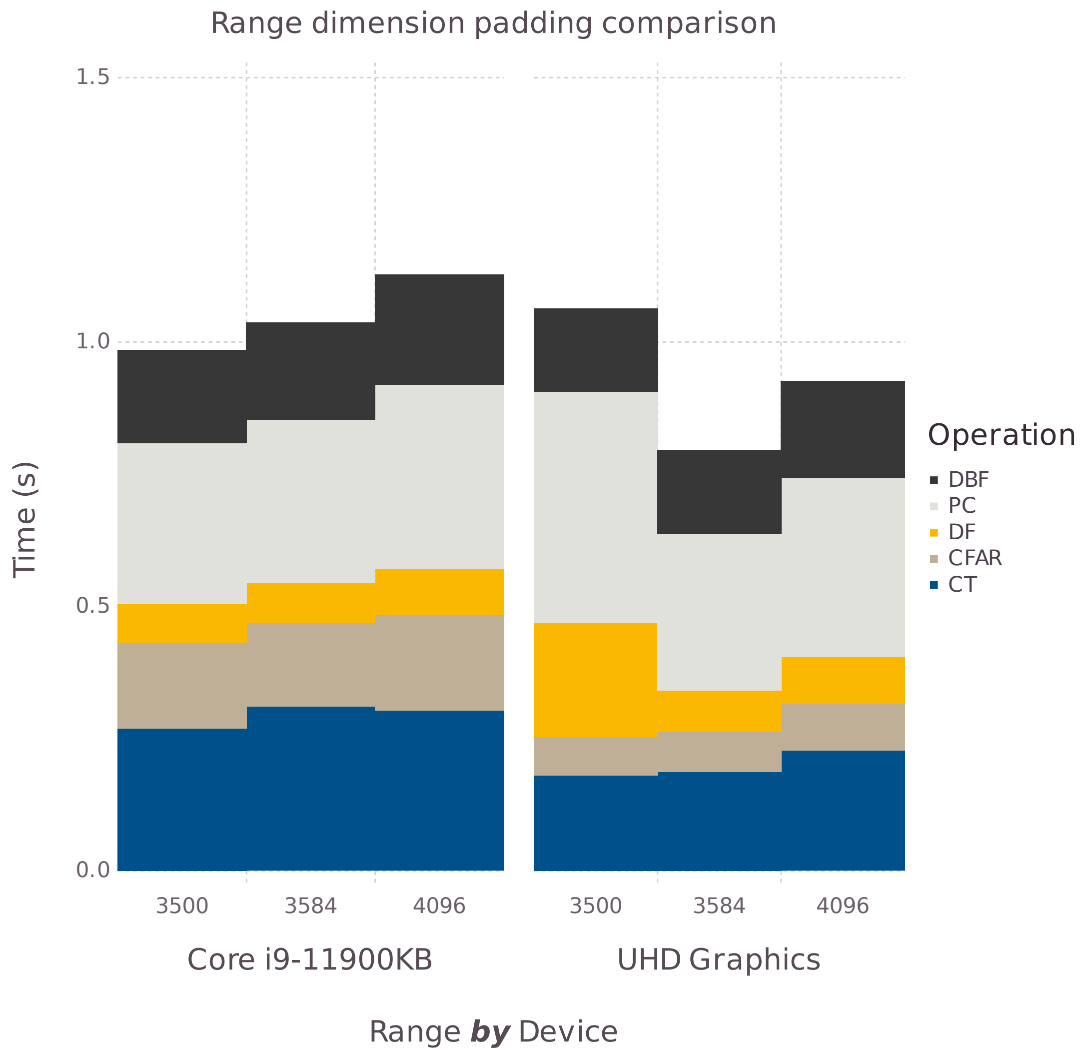

Figure 7.3: Timings for 10 iterations of the full pipeline with different range dimension padding and devices.

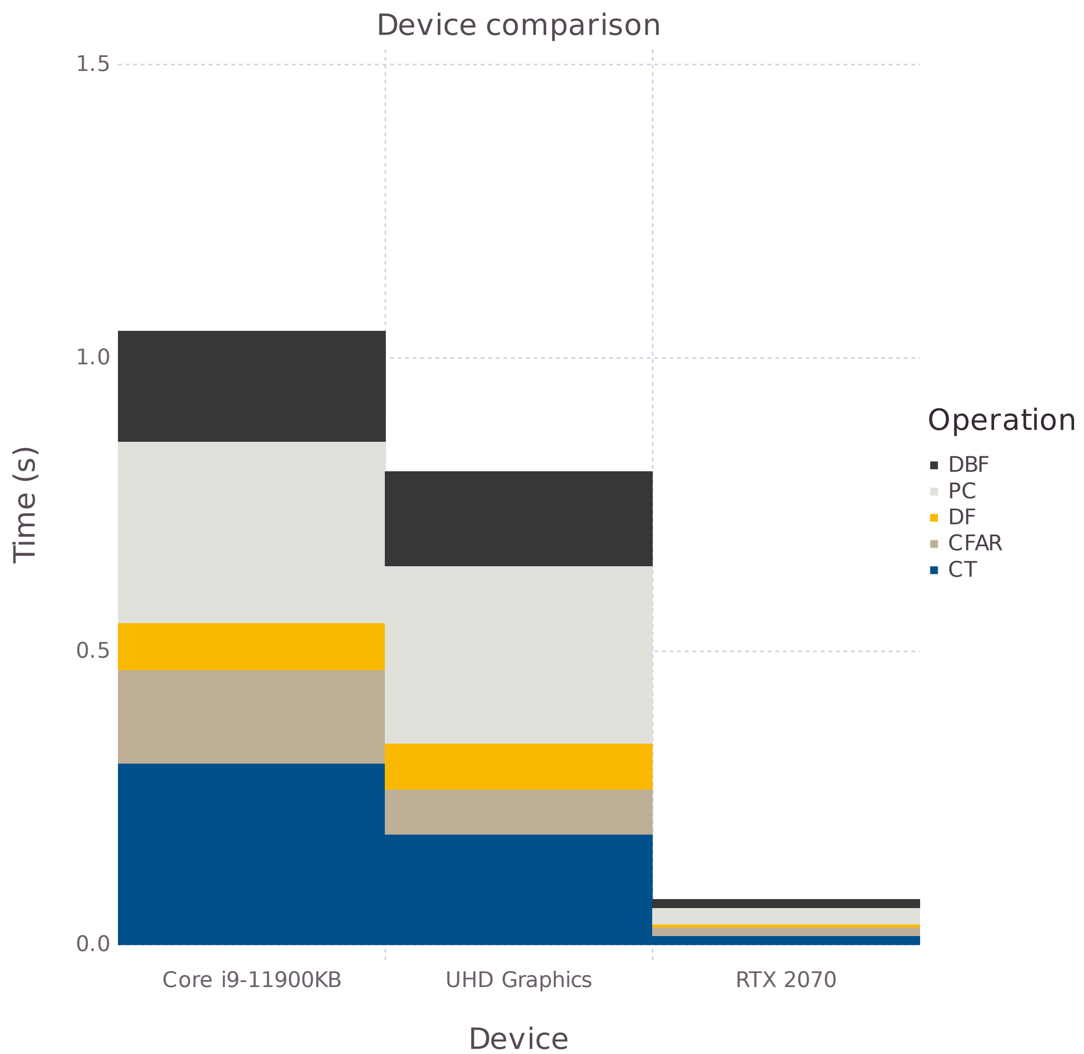

Figure 7.4: Timings for 10 iterations of the full pipeline with range dimension = 3584 and different devices.

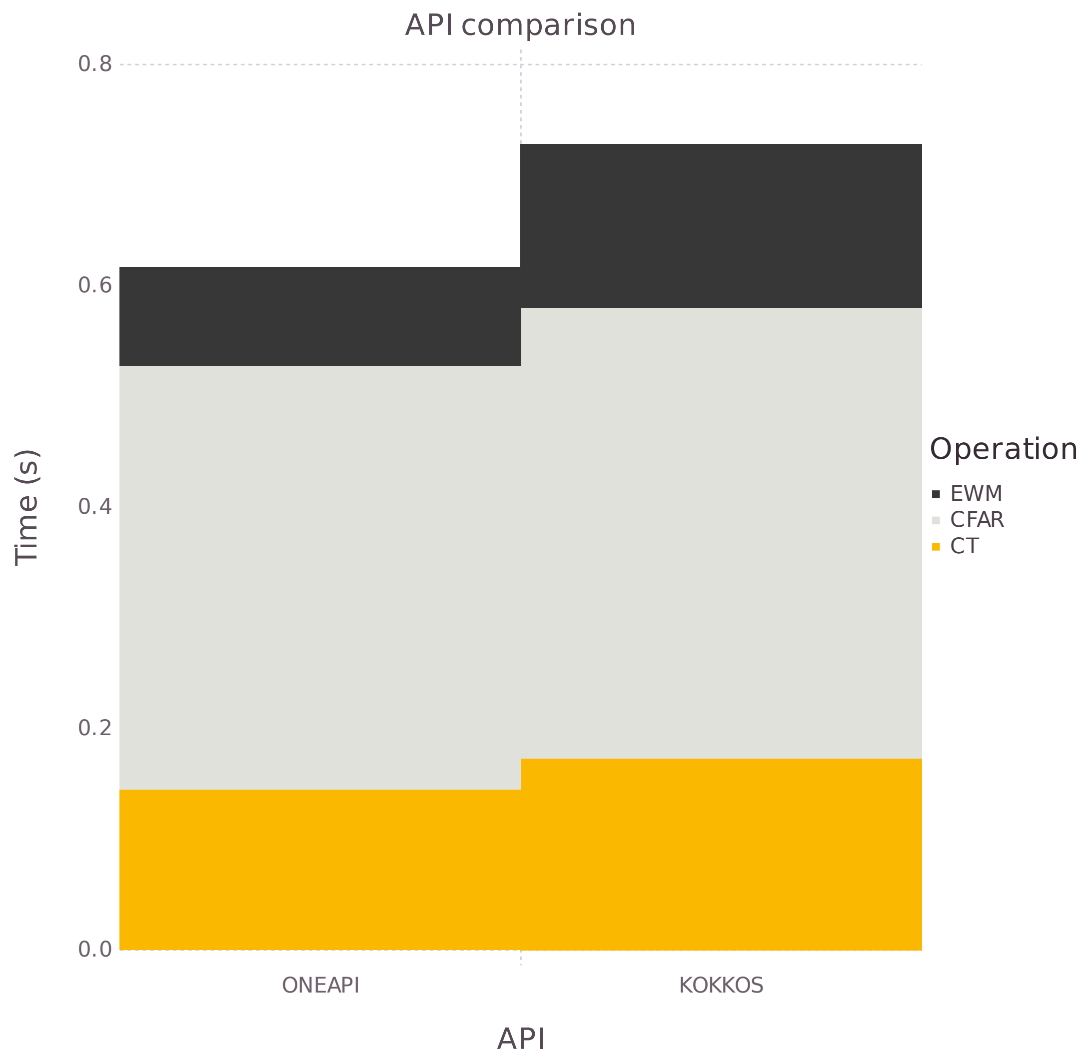

Figure 7.5: Timings for 10 iterations of kernels on CPU implemented with oneAPI, Kokkos.

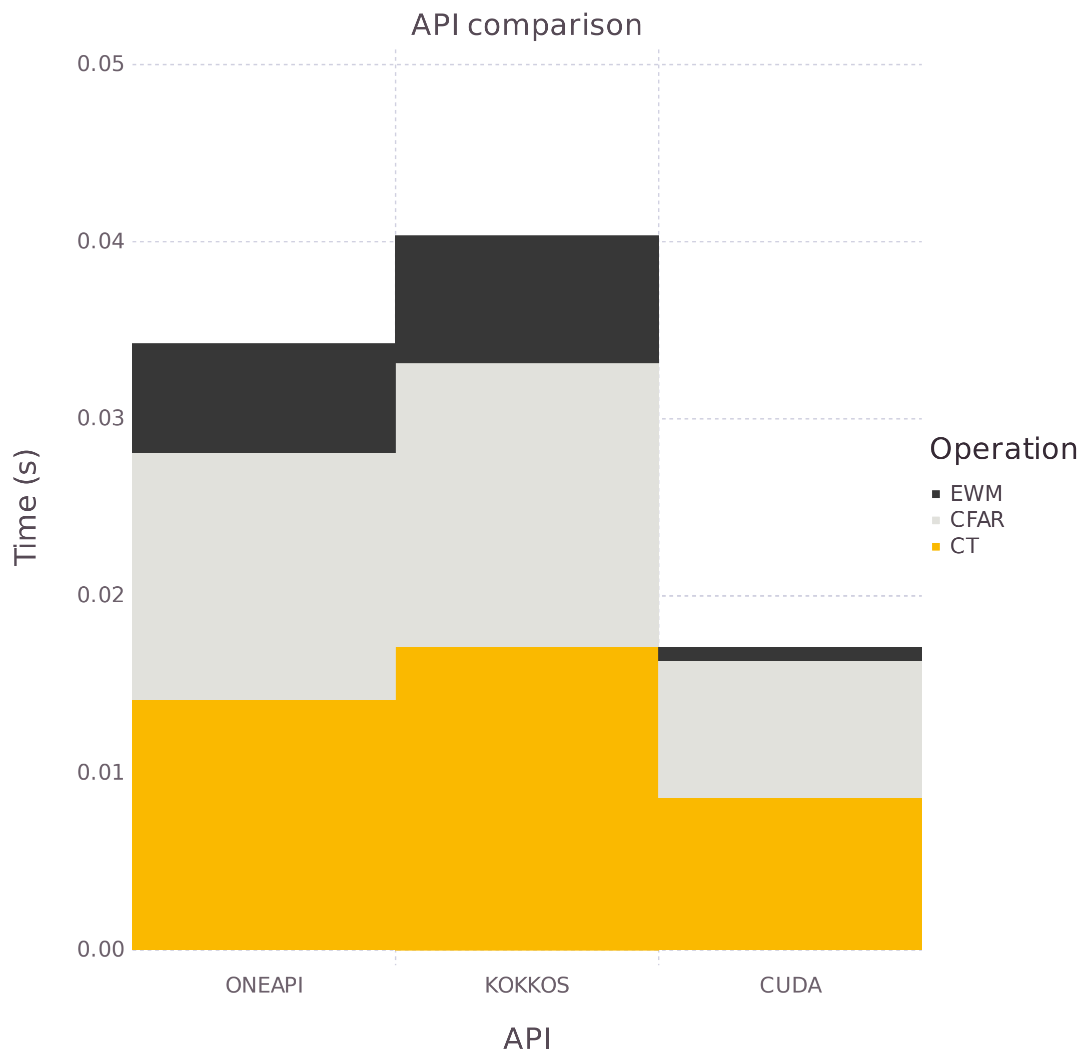

Figure 7.6: Timings for 10 iterations of kernels on GPU implemented with oneAPI, Kokkos and CUDA.

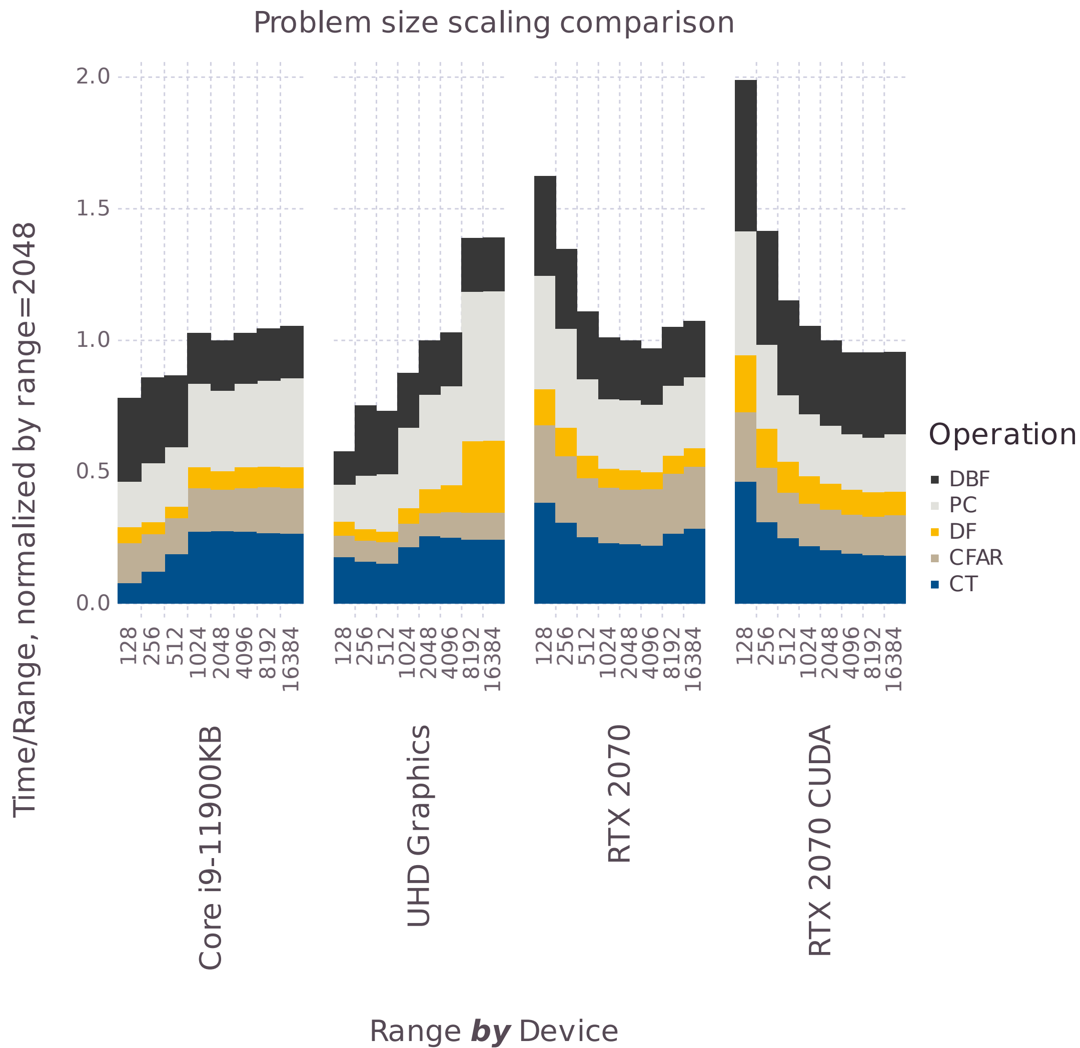

Figure 7.7: Timings for different ranges and devices. The y-axis shows the time divided by range and normalized by range = 2048 for each device. The first three devices use oneAPI, while RTX 2070 CUDA uses kernels written in CUDA.

Chapter 8
Discussion

In this chapter we will summarize and discuss the main findings and try to answer the research questions. Then, we will discuss limitations of the study and future work.

8.1 Interpretation of Results

8.1.1 Productivity

The research question about productivity asked how much code was needed in comparison to similar programming models. Our results show that DPC++ has one advantage compared to Kokkos and CUDA, which is the automatic work group size. Otherwise, it is simpler to submit kernels in both Kokkos and CUDA, and Kokkos has useful data wrapper that can further reduce the amount of code needed to implement kernels.

While the productivity is not necessarily better than the other two programming models, we also do not consider there to be a large enough difference to sway developers in either direction, without considering other aspects.

8.1.2 Portability

Across the performance experiments, we can see that oneAPI does successfully compile and execute on three different architectures and four different devices: an Intel Core i9-11900KB CPU, an AMD Ryzen 5 3600 CPU, an Intel UHD Graphics integrated GPU, and an Nvidia Geforce RTX 2070 GPU. While we did manage to compile for an Intel Stratix 10 SX FPGA after finding a workaround for the incorrect Intel documentation and server setup,the code did not successfully execute. Further details about the oneAPI setup process can be found in Appendix A. The lack of oneMKL support on the Intel FPGA is also disappointing, given that it is a first-party platform.

Although we got the code to execute on a few different devices, the code did require changes to run well. A potential bug with the inverse FFT being much slower on CPU when reusing FFT plans, while for GPUs, creating new plans for each iteration added significant overhead. Therefore, we had to reuse FFT plans on GPUs but not CPUs to get good performance. Another potential bug made the automatic work group size unreliable, so we tuned the work group size manually for each device. Finally, we had to use the cuFFT library directly, instead of through the oneMKL interface, due to the unfinished state of its Nvidia GPU support. The batched general matrix multiplication function, on the other hand, was available for Nvidia GPUs.

Fixing the FFT issue, so that it is always preferable to reuse FFT plans, when possible, would be a step towards better portability. Another step would be improving the automatic work group size feature, so that it is more reliable and closer to the performance of manual tuning. Intel could also implement oneMKL for their FPGAs, so the code is at least portable among its first-party hardware.

The larger problem is third-party adoption. As long as Nvidia is still dominant in the GPU market and people develop for CUDA, Nvidia has no good reason spending resources on implementing the cross-architecture oneAPI. Rather, it would be harmful to Nvidia, as it makes it easier for developers to move away from their platform. On the other hand, once there is broad support for oneAPI, the API should become appealing for hardware producers that are not already dominant with lot of software written for their platform. Then, they could potentially get support for all of that software just by implementing oneAPI for its hardware. But, until oneAPI reaches that amount of support, Intel will likely have to spend those resources itself.

In its current state, oneAPI does provide good portability between Intel CPUs and GPUs, but in our experience, not for FPGAs. Kernels work on Nvidia GPUs and there is limited oneMKL support. In summary, the portability of oneAPI is an improvement from vendor specific programming models like CUDA, but it has not yet reached its goals of being a broadly supported industry standard.

8.1.3 Performance

The baseline comparison in Section 7.1 showed that oneAPI is competitive with and even exceeded the performance of popular math libraries Eigen and FFTW used for signal processing. The fact that the oneAPI FFTs were slightly faster is not surprising given that the oneMKL implementation is developed by Intel, with much more resources, than the researchers at MIT responsible for the FFTW implementation. FFTW is also portable, while the Intel's oneMKL implementation is tuned exclusively for its own hardware.

In Section 7.2, the results showed that letting the implementation choose work group size can get close to manually tuned performance, but in certain cases fails. In the experiment, one of the corner turns got about the same performance as setting the work group size to one. That is the worst scenario for architectures, such as GPUs, that rely heavily on parallelism, since it masks out all SIMD units except for one. Currently, the automatic work group size feature seems unreliable, especially since its effectiveness is dependent on input size. Unless only one specific input size will be used in the application, it is difficult to know whether the problem will show up for other input sizes.

The CPU vs integrated GPU comparison in Section 7.3 showed that Intel's GPU imple-

mentation of FFT is much more sensitive to good size factors than the CPU implementation and that offloading to the integrated GPU can improve performance. However, the sensitive FFT implementation on Intel GPUs means that achieving maximum performance requires proper padding. It is also important to reuse the FFT plans, since they are slow to create for the GPU.

Comparing with the discrete Nvidia GPU in Section 7.4, we can see that the performance scales well to larger third-party GPUs.

In Section 7.5 and Section 7.6 the results showed that DPC++ kernels consistently outperform Kokkos on both CPU and GPU. On the Nvidia GPU, CUDA is at much faster for all three operations, which indicates that the portability comes with a performance cost for third-party hardware. DPC++ performing worse than CUDA on Nvidia GPUs is also corroborated by some of the papers mentioned in Chapter 4. Given that the elementwise multiplication is nearly eight times faster in CUDA than DPC++, the gained portability does not seem like a very good trade-off when running on Nvidia GPUs. However, since the oneMKL interface wraps native libraries such as cuBLAS, those functions will have native performance. Thus, if an application mostly use oneAPI library functions and only have a few DPC++ kernel implementations, the performance loss will be less substantial.

The aim of the final experiment in Section 7.7, was to measure how well performance scales to different problem sizes. Time complexity for pulse compression, when scaling the range dimension, is $O(n \log(n))$ due to the FFTs operating on the range dimension, while it is $O(n)$ rest of the operations. In Figure 7.7, we divide the time by the range dimension. Therefore, with perfect scaling, pulse compression is expected to increase in a logarithmic fashion, while the other operations should stay constant.

As mentioned in Section 5.4, the Nvidia GPU is expected to have additional kernel launch and synchronization overhead. From the results it is apparent that the Nvidia GPU needs large problem sizes to get fully utilized and to amortize kernel launch and synchronization latencies. As with the kernel comparison, DPC++ also seems to scale worse than CUDA, since its performance starts getting worse at smaller problem sizes than CUDA.

The CPU performances scales well to small problem sizes compared to the Nvidia GPU, due to less overhead. At the smallest sizes, it also benefits from being able to keep all data in cache, especially for memory bandwidth intensive operations like corner turns. More surprising is that the integrated Intel GPU also performs best at small problem sizes, given the overhead issues of the Nvidia GPU. On the other hand, being integrated naturally lends itself to low latency communication. The fact that it is sharing last level cache and global memory with the host CPU should also eliminate the memory transfer overhead. Considering the low overhead, offloading even single operations to an integrated GPU is useful.

8.2 Broader Impact

The primary benefit of using oneAPI is that it does not lock the software to a single hardware platform. As mentioned in Section 8.1.2, Nvidia wants people to use CUDA, so that they have to keep using Nvidia hardware, unless they rewrite all their code. Even if oneAPI does not support that many platforms currently, the fact that the standard is open and free for any company to adopt makes it more likely that it will support more platforms in the future than platform specific programming models like CUDA. With SYCL kernels, developers are not

even locked to using an Intel DPC++ compiler, as any SYCL implementation can be used.

With platform specific code, decisions have to be made about which platform is currently most suitable, with the hope that it will continue to be the leading platform in the future. If it turns out another platform becomes superior it can be expensive to switch, given that all code has to be ported over to the new platform. That cost could prompt the decision to stay with the inferior platform. Reduced vendor lock-in makes competition stronger, since developers can use the cross-architecture code with whichever vendor produces the best hardware at any given point. An example of this is the x86 CPU market where competition between AMD and Intel is strong thanks to the shared code base, whereas in the GPU compute market, Nvidia dominates due to its greater CUDA adoption [32] which hinders usage of Intel and AMD GPUs.

An open specification such as oneAPI also makes it easier for new hardware providers to join the market and provide more choice and competition, since the code will work for any hardware that supports the specification. Intel recently announced the oneAPI Construction Kit [38] that is supposed to make it simpler for third-parties to implement oneAPI for their custom architecture.

However, given oneAPI's current portability limitations, it does not yet fulfill its lofty goals of becoming a fully unified programming model that can run on any processor.

8.3 Limitations

The results of this study might not apply to all situations. The range of algorithms tested is limited and focused on simple linear algebra. For example, the experiments do not test data parallel algorithms like reductions and scans. However, the operations considered, such as matrix multiplication, FFT and convolution, are very common and used in popular fields like signal processing and machine learning.

8.4 Future Work

Unfortunately, this thesis was not able to investigate the performance of oneAPI on FPGAs. However, the difficulties experienced do give insight in portability and productivity. It was not as simple to switch to FPGA as switching between CPU and GPU. As with standard FPGA development compile times are long and Intel DevCloud does not support simulation, which otherwise could have enabled shorter iteration times. Further experiments on FPGA would be interesting as future work. It would also be interesting to investigate what what makes oneAPI faster than Kokkos when both use the SYCL backend and why their kernels are much slower than CUDA on Nvidia GPUs.

Chapter 9
Conclusion

This master's thesis has evaluated the productivity, portability and performance of oneAPI, a unified programming model developed by Intel, in the context of radar signal processing applications. This evaluation was necessitated by the high-performance requirements of active electronically scanned array (AESA) radar systems, which can benefit from the use of highly parallel hardware such as GPUs and FPGAs. Traditional programming models present challenges when upgrading hardware or using multiple hardware types due to a lack of code portability.

Productivity: The study found that while DPC++ does not offer superior productivity to other programming models such as Kokkos and CUDA. The differences are not significant enough to clearly favor one model over another. Automatic work group size in DPC++ improves productivity, but is unreliable. Moreover, certain functionalities such as kernel submission are simpler in Kokkos and CUDA.

Portability: With respect to portability, oneAPI was successfully able to compile and execute across multiple architectures and devices, including both Intel and AMD CPUs, as well as Nvidia GPUs. However, issues were encountered in attempting to use it with an Intel FPGA, and several bugs impacted the performance across different devices. In its current state, oneAPI shows good portability between Intel CPUs and GPUs, but there are limitations in third-party and FPGA support.

Performance: OneAPI demonstrated competitive performance with other popular math libraries used for signal processing, like Eigen and FFTW. However, it was found that oneAPI's DPC++ performs worse than CUDA on Nvidia GPUs, indicating a performance cost for third-party hardware. This suggests that while oneAPI can offer improvements in portability, it may do so at the expense of performance. Despite this, if an application mostly uses oneAPI library functions and only has a few DPC++ kernel implementations, the performance loss will be less substantial.

References

[1] Saab AB. Saab's fourth globaleye conducted successful first flight. https://www.saab.com/newsroom/press-releases/2023/saabs-fourth-globaleye-conducted-successful-first-flight, 2023. Accessed 08-06-2023.

[2] A. Ahlander, A. Åström, B. Svensson, and M. Taveniku. Meeting engineer efficiency requirements in highly parallel signal processing by using platforms. In *IASTED PDCS*, pages 693–700, 2005.

[3] J. I. Aliaga, R. Reyes, and M. Goli. Sycl-blas: Leveraging expression trees for linear algebra. In *Proceedings of the 5th International Workshop on OpenCL*, IWOCL 2017, New York, NY, USA, 2017. Association for Computing Machinery.

[4] A. D. Brown. *Active Electronically Scanned Arrays: Fundamentals and Applications*. Wiley-IEEE Press, 2021.

[5] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[6] Intel Corporation. Intel advisor. https://www.intel.com/content/www/us/en/developer/tools/oneapi/a 2023. Accessed 19-06-2023.

[7] Intel Corporation. Intel iris xe gpu architecture. https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-0/intel-iris-xe-gpu-architecture.html, 2023. Accessed 19-06-2023.

[8] Intel Corporation. Intel oneapi documentation. https://www.intel.com/content/www/us/en/developer/tools/oneapi/documentation.html, 2023. Accessed 08-06-2023.

[9] Intel Corporation. Intel oneapi dpc++/c++ compiler developer guide and reference. https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2023-1/overview.html, 2023. Accessed 21-06-2023.

[10] Intel Corporation. Intel oneapi level zero. https://www.intel.com/content/www/us/en/developer/tools/oneap 2023. Accessed 21-06-2023.

[11] Intel Corporation. Intel oneapi level zero. https://www.intel.com/content/www/us/en/developer/tools/oneap 2023. Accessed 21-06-2023.

[12] Intel Corporation. Intel oneapi level zero. https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2023-1/intel-oneapi-level-zero.html, 2023. Accessed 21-06-2023.

[13] Intel Corporation. Intel oneapi math kernel library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html, 2023. Accessed 19-06-2023.

[14] Intel Corporation. Intel vtune profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/v profiler.html, 2023. Accessed 19-06-2023.

[15] Intel Corporation. oneapi math kernel library (onemkl) interfaces. https://github.com/oneapi-src/oneMKL, 2023. Accessed 19-06-2023.

[16] Intel Corporation. oneapi specification documentation. https://spec.oneapi.io/versions/latest/introduction.html, 2023. Accessed 08-06-2023.

[17] Nvidia Corporation. cublas api reference. https://docs.nvidia.com/cuda/cublas/, 2023. Accessed 25-06-2023.

[18] Nvidia Corporation. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/contents.html, 2023. Accessed 27-05-2023.

[19] Nvidia Corporation. cufft api reference. https://docs.nvidia.com/cuda/cufft/index.html, 2023. Accessed 25-06-2023.

[20] Nvidia Corporation. Nvidia cuda compiler driver nvcc. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html, 2023. Accessed 19-06-2023.

[21] M. Costanzo, E. Rucci, C. García Sánchez, and M. R. Naiouf. Early experiences migrating CUDA codes to oneapi. *CoRR*, abs/2105.13489, 2021.

[22] Advanced Micro Devices. Introduction to hip programming guide. https://docs.amd.com/bundle/HIP-Programming-Guide-v5.3/page/Introduction_to_HIP_Programming_Guide.html/, 2023. Accessed 21-06-2023.

[23] Advanced Micro Devices. Programming an fpga: An introduction to how it works. https://www.xilinx.com/products/silicon-devices/resources/programming-an-fpga-an-introduction-to-how-it-works.html, 2023. Accessed 21-06-2023.

[24] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[25] M. Frigo and S. G. Johnson. Fftw 3.3.10. https://www.fftw.org/fftw3_doc/index.html, 2023. Accessed 19-06-2023.

[26] The Khronos SYCL Working Group. Sycl 2020 specification. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html, 2023. Accessed 08-06-2023.

[27] The Khronus Group. Opencl. https://www.khronos.org/opencl/, 2023. Accessed 21-06-2023.

[28] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010. Accessed 27-05-2023.

[29] Johnny's Software Lab. Crash course introduction to parallelism: Simd parallelism. https://johnnysswlab.com/crash-course-introduction-to-parallelism-simd-parallelism/, 2021. Accessed 21-06-2023.

[30] J. Lebak, J. McMahon, and M. Arakawa. Polymorphous computing architecture (pca) application benchmark 1: Three-dimensional radar data processing. page 22, 11 2001.

[31] E. Marinelli and R. Appuswamy. Xjoin: Portable, parallel hash join across diverse XPU architectures with oneapi. In D. Porobic and S. Blanas, editors, *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China*, pages 11:1–11:5. ACM, 2021.

[32] S. McIntosh-Smith. Large scale hpc hardware in the age of ai. https://www.iwocl.org/wp-content/uploads/iwocl-2023-Simon-Mcintosh-Smith-opening.pdf, 2023. Accessed 22-06-2023.

[33] National Technology & Engineering Solutions of Sandia. Kokkos: The programming model. https://kokkos.github.io/kokkos-core-wiki/, 2023. Accessed 27-05-2023.

[34] J R. Reinders. Intel avx-512 instructions. https://www.intel.com/content/www/us/en/developer/articles avx-512-instructions.html, 2017. Accessed 21-06-2023.

[35] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian. *Data Parallel C++*. Apress, 2021.

[36] R. Reyes and V. Lomüller. SYCL: single-source C++ accelerator programming. In G. R. Joubert, H. Leather, M. Parsons, F. J. Peters, and M. Sawyer, editors, *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK*, volume 27 of *Advances in Parallel Computing*, pages 673–682. IOS Press, 2015.

[37] M.I. Skolnik. *Introduction to Radar Systems*. Electrical engineering series. McGraw-Hill, 2001.

[38] Codeplay Software. Software first with the oneapi construction kit. https://codeplay.com/portal/press-releases/2023/06/05/software-first-with-the-oneapi-construction-kit, 2023. Accessed 08-06-2023.

[39] J. Stokes. Introduction to multithreading, superthreading and hyperthreading. https://arstechnica.com/features/2002/10/hyperthreading/, 2002. Accessed 21-06-2023.

[40] V. Strassen. *Gaussian Elimination is not Optimal*, volume 13. Springer, 1969.

[41] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817, 2022.

[42] V. Volokitin, A. V. Bashinov, E. Efimenko, A. A. Gonoskov, and I. B. Meyerov. High performance implementation of boris particle pusher on DPC++. A first look at oneapi. In Victor Malyshkin, editor, *Parallel Computing Technologies - 16th International Conference, PaCT 2021, Kaliningrad, Russia, September 13-18, 2021, Proceedings*, volume 12942 of *Lecture Notes in Computer Science*, pages 288–300. Springer, 2021.

[43] R. W. Vuduc and J. Choi. A brief history and introduction to gpgpu. 2013.

[44] Y. Wang, Y. Zhou, Q. Scott Wang, Y. Wang, Q. Xu, C. Wang, B. Peng, Z Zhu, K. Takuya, and D. Wang. Developing medical ultrasound beamforming application on GPU and FPGA using oneapi. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021, Portland, OR, USA, June 17-21, 2021*, pages 360–370. IEEE, 2021.

Appendices

Appendix A
oneAPI Setup Process

A.1 Intel Hardware

Using oneAPI requires installing the compiler, libraries and setting up environment variables. Intel provides official packages for popular Linux distributions like Ubuntu, Red Hat, Fedora and SUSE. Arch Linux is not officially supported, but still provides a package in the community repository.

Before compiling, the environment variables can be configured by sourcing the included `setvars.sh` script.

```
. /opt/intel/oneapi/setvars.sh
```

An alternative is using an official Docker image, like `intel/oneapi-basekit`, which sets everything up for you.

The recommended way of integrating oneAPI libraries is using CMake and the corresponding included config files, such as `MKLConfig.cmake` for oneMKL. A small example might look like this:

```
cmake_minimum_required(VERSION 3.13)
project(oneMKL_program LANGUAGES CXX)
find_package(MKL CONFIG REQUIRED)
add_executable(myprogram, myprogram.cpp)

target_compile_options(myprogram
    PUBLIC $<TARGET_PROPERTY:MKL::MKL,INTERFACE_COMPILE_OPTIONS>)
target_include_directories(myprogram
    PUBLIC $<TARGET_PROPERTY:MKL::MKL,INTERFACE_INCLUDE_DIRECTORIES>)
target_link_libraries(myprogram
    PUBLIC $<LINK_ONLY:MKL::MKL>)
```

A.2 FPGA

FPGA is one of the hardware platform DPC++ supports. Given the high performance and efficiency achievable with FPGA thanks to pipelining and parallelization, it is an interesting platform for radar signal processing. However, running the code on an FPGA in the Intel Devcloud is not as easy as just changing the SYCL queue device.

The first step is compiling the code for FPGA hardware. Note that the documentation for how to compile for FPGA is either incorrect or the Intel Devcloud server setup is broken, as the suggested way is not working at the time of writing this. Specifically, the documentation tells you to use `fpga_compile` nodes for compiling and `fpga_runtime` to run. However, the only way we could get compilation for FPGA hardware to work was to also compile on an `fpga_runtime` node. The easiest way to access an `fpga_runtime` node is to login to Intel Devcloud with `ssh` and run

```
. /data/intel_fpga/devcloudLoginToolSetup.sh
devcloud_login
```

Choose one of the oneAPI options. The next problem is that not all of the nodes have CMake 3.13 installed, which is needed for the CMake config above. As a workaround you can create a *Conda* environment and install a more recent version there.

```
conda create -n build python=3.11
conda activate build
conda install -c anaconda cmake
```

When compiling for an Intel FPGA, you have to add the `-fintelfpga` and `-Xshardware` options and specify a target architecture with `-Xstarget`. The `Xs` prefix means the argument is passed to the FPGA backend. Here is an example where the target is an Intel Stratix 10 SX with both explicit and restricted USM support. Restricted USM means it also allows for shared USM allocations.

```
icpx -fsycl -fintelfpga -Xshardware \
    -Xstarget=intel_s10sx_pac:pac_s10_usm myprogram.cpp
```

Finally, if the code uses USM, the FPGA board has to be explicitly initialized. For example, if the code is compiled with: The board must be initialized with:

```
aocl initialize acl0 pac_s10_usm
```

There are also some other flags such as `-Xsemulator` and `-Xssimulation` which compiles for emulation and simulation, respectively. However, only emulation is supported in Intel Devcloud. Compiling for FPGA hardware can take several hours, so the emulator is useful to test with first.

Although the oneAPI specification website [16] says

> "As part of oneAPI, oneMKL is designed to allow execution on a wide variety of computational devices: CPUs, GPUs, FPGAs, and other accelerators."

, oneMKL does not currently seem to support FPGA. Neither general matrix multiplication nor FFT is available on the Intel Stratix 10 SX.

A.3 Nvidia GPUs

Intel's newly acquired subsidiary Codeplay provides Nvidia GPU support for oneAPI. It requires installing an extension that is downloaded from their website. The SYCL program can then be compiled with:

```
clang++ -fsycl -fsycl-targets=nvptx64-nvidia-cuda \
    myprogram.cpp -o myprogram
```

To use the open-source oneMKL interfaces, you have to build and install the library with the CUDA backend enabled. The BLAS functions can then be linked with using `-lonemkl_blas_cublas`. At the time of writing, it does not have complete BLAS support and no DFT interfaces for Nvidia GPUs.

EXAMENSARBETE Evaluating a Unified Parallel Computing API for Radar Signal Processing
STUDENT Filip Jergle Almquist
HANDLEDARE Anders Åhlander och Jonas Skeppstedt (LTH)
EXAMINATOR Flavius Gruian (LTH)

Återanvändbar programmeringsmodell för ökad hållbarhet och effektivitet

POPULÄRVETENSKAPLIG SAMMANFATTNING **Filip Jergle Almquist**

Detta examensarbete undersöker potentialen av oneAPI för att effektivisera utveckling and öka återanvändning av kod för radarsignalbehandling. Resultaten indikerar att oneAPI-kod kan köras på flera platformar, men har en del begränsningar.

Radar är viktiga verktyg som används inom flera områden, till exempel inom försvar, luftfartsledning och väderövervakning. Nya tekniker som AESA (Active Electronically Scanned Array) möjliggör dock ännu mer flexibla radarlösningar med högre upplösning och bättre prestanda. De ställer dock höga krav på datorkraft eftersom stora mängder data måste bearbetas i realtid. Därför uttnyttjas flera olika typer av hårdvara, både traditionella datorprocessorer och specialiserade processorer som grafikprocessorer och FPGA-kretsar. Utmaningen är att programmeringen för dessa processorer kan variera kraftigt, vilket gör att stora delar av koden måste skrivas om när typen av hårdvara byts ut.

Detta är ett problem inom radartekniken, där effektiv programmering är avgörande och system ofta är i bruk i flera decennier. För att lösa detta undersöker Saabs radaravdelning möjligheten att använda en universell programmeringsmodell som kallas oneAPI. oneAPI skapades av Intel och syftar till att göra det möjligt att återanvända samma kod oavsett vilken typ of processor man använder.

I mitt examensarbete har jag utvärderat oneAPI:s potential för att hantera den krävande signalbehandlingen som krävs för radarapplikationer i de tre olika aspekterna produktivitet, portabilitet och prestanda. När det gäller produktivitet visar resultaten att oneAPI likvärdig med liknande programmeringsmodeller. oneAPI-koden kan återanvändas i hög grad mellan olika typer av hårdvara, men vissa delar av oneAPI är fortfarande under utveckling, så stödet sig kraftigt mellan olika processortyper, vilket begränsar portabiliteten. Prestandamässigt gör oneAPI starkt i från sig, framförallt på Intels egna hårdvara, och producerar snabbare kod än andra etablerade programmeringsmodeller. På hårdvara från en konkurrent visar resultaten däremot att prestandan kan vara upp till åtta gånger långsammare, vilket tyder på att den ökade portabiliteten av den universella programmeringsmodellen kan ha en stor prestandakostnad jämfört med platformsspecifik kod.

I nuläget lever oneAPI bara delvis upp till dess mål som universell programmeringsmodel och det återstår att se hur väl denna teknik kommer att antas av andra företag inom industrin och hur den kommer att utvecklas framöver.